Sixth Generation Computers

Dr. Dobb's Journal

For the Experienced in Microcomputing

#91 MAY 1984
$2.95 (3.50 Canada)

A New Library for Small C

Introduction to Modula 2

Converting Fig-Forth to Forth-83

0 38351 16562 05

Self-dFENSE™ for EDP managers.

The micro invasion has begun. And, chances are, you've now got a lot of different people in a lot of different departments using a lot of different micros.

Now there's a way for you to control and maximize the benefits of all the different micros in your domain.

Fight back with dBASE II.®

dBASE II is the relational database management system from Ashton-Tate that enables you to manage your micro-based corporate data resources with the high level of consistency and sophistication you've enjoyed with mainframe and minicomputer systems.

Armed with dBASE II and the dBASE II RunTime™ program development module, you can write programs which will enable micro users in each department to "do their own thing" while creating complete database consistency throughout the company.

dBASE II is a powerful, flexible way for you to effectively manage the micro proliferation.

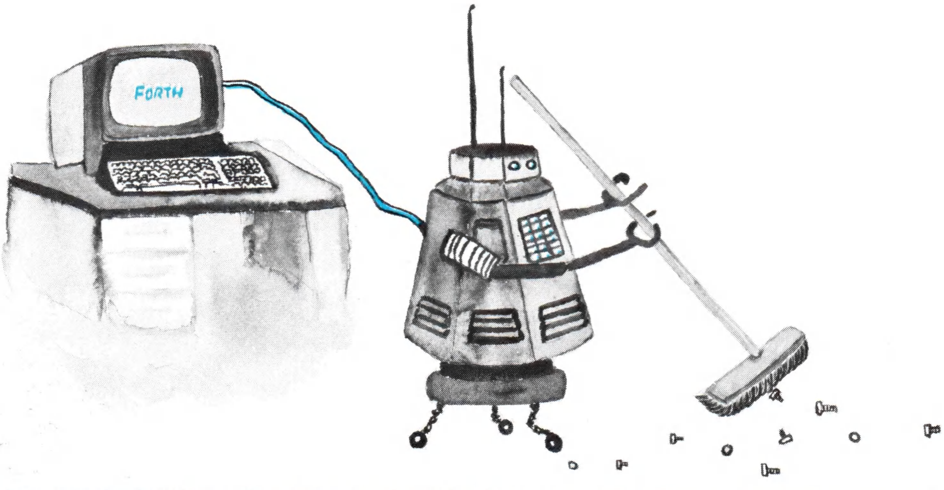
Help is here.

If you'd like to know more about how dBASE II and RunTime can help you win the micro management battle, contact Ashton-Tate today. 10150 West Jefferson Boulevard, Culver City, CA 90230. (800) 437-4329, ext. 217 . In Colorado (303) 799-4900. In the U.K. (0908) 568866.

ASHTON·TATE™

Circle **no. 1** on reader service card.

FORTH GIVES YOU TOTAL CONTROL



GRAPHICS • GAMES • COMMUNICATIONS
ROBOTICS • DATA ACQUISITION • PROCESS CONTROL

FORTH: for Z-80®, 8080, 8086, 68000, and IBM® PC
(Complies with the New 83-Standard)

- ***FORTH* programs are instantly portable across the four most popular microprocessors.**
- ***FORTH* is interactive and 20 times faster than BASIC.**
- ***FORTH* programs are highly structured and easy to maintain.**
- ***FORTH* provides direct control over all interrupts, memory locations, and i/o ports.**
- ***FORTH* allows full access to DOS files and functions.**
- ***FORTH* application programs can be distributed as turnkey COM files with no license fee.**
- ***FORTH* Cross Compilers are available for ROM'ed or disk based applications on most microprocessors.**
- **Custom programming, consulting, and educational services available.**

FORTH Application Development Systems include interpreter/compiler with virtual memory management and multitasking, assembler, full screen editor, decompiler, utilities, and detailed technical manual. Standard random access files used for screen storage, extensions provided for access to all operating system functions.

Z-80 FORTH for CP/M® 2.2 or MP/M II, $50.00; **8080 FORTH** for CP/M 2.2 or MP/M II, $50.00; **8086 FORTH** for CP/M-86 or MS-DOS, $100.00; **PC/FORTH™** for PC-DOS, CP/M-86, or CCPM, $100.00; **68000 FORTH** for CP/M-68K, $250.00

FORTH + Systems are 32 bit implementations that allow creation of programs as large as 1 megabyte. The entire memory address space of the 68000 or 8086/88 is supported directly for programs and data.

PC FORTH + ... $250.00
8086 FORTH + for CP/M-86 . . . . . . $250.00
68000 FORTH + for CP/M-68K . . . $400.00

FORTH Cross Compiler allows you to customize the FORTH nucleus, recompile on a host computer for a different target computer, generate headerless and ROM-able code. Supports forward referencing. Produces executable image in RAM or disk file. No license fee for applications, $300.00.

FORTH Native Code Compilers
For Z80 FORTH and CP/M $100.00
For 8086 FORTH and CP/M-86 . . . . . $200.00
For IBM PC and PC-DOS $200.00

Extension Packages
Software floating point (Z-80, 8086, PC only), $100.00; AMD 9511 support (Z-80, 8086, 68000 only), $100.00; Intel 8087 support (8086, PC only), $100.00; Advanced color graphics (PC only), $100.00; Symbolic interactive debugger (PC only), $100.00; PC/TERM Communications/file transfer for Smartmodem, $60.00; Cross reference utility, $25.00; PC/GEN (custom character sets, PC only), $50.00; Curry FORTH Programming Aids, $150.00; B-Tree index manager, $125.00; B-Tree index and file manager, $200.00; QTF + Screen editor for IBM PC, $100.00; Quad Integer Math Pack, $25.00.

AUGUSTA, Ada subset compiler from Computer Linguistics for Z-80 CP/M 2.2 systems, $90.00

"Starting FORTH" tutorial by Brodie, softcover, $16.00.

INTEL 8087-3 Numeric Coprocessor, $250.00

Z-80 and 8080 *FORTH* require 48 Kbytes RAM. 8086 and 68000 *FORTH* require 64 Kbytes RAM. Disk formats available include: 8" standard CP/M SSSD, Northstar 5¼" QD, Kaypro 5¼", Apple 5¼", Micro-Mate 5¼", MS-DOS 5¼", Osborne 5¼", DD, and Sage. Most other formats can be special ordered. Dealer inquiries invited.

Z-80 is a registered trademark of Zilog, Inc.; CP/M is a registered trademark of Digital Research, Inc.; IBM is a registered trademark of International Business Machines Corp.; Augusta is a trademark of Computer Linguistics; PC/FORTH and PC/GEN are trademarks of Laboratory Microsystems Inc.

LMI **Laboratory Microsystems Incorporated**
4147 Beethoven Street, Los Angeles, CA 90066
Phone credit card orders to ***(213) 306-7412***

VISA® MasterCard

Circle **no. 33** on reader service card.

Circle **no. 36** on reader service card.

Dr. Dobb's Journal

For the Experienced in Microcomputing

May 1984
Volume 9, Issue 5

CONTENTS

ARTICLES

DEPARTMENTS

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto CA 94303, (415) 424-0600, *under license from People's Computer Company*, Box E, Menlo Park, CA 94626. Second class postage paid at Palo Alto and at additional entry points. Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, 2464 Embarcadero Way, Palo Alto CA 94303.

Subscription Rates: $25 per year within the United States $44 for first class to Canada and Mexico, $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Contributing Subscribers: Christine Bell, W. D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor Jr., Ted A. Reuss III, InfoWorld, Stan Veit, Western Material Control, S. P. Kennedy, John Hatch, Richard Jorgensen, John Boak, Bill Spees, R. B. Sutton. **Lifetime Subscribers:** Michael S. Zick, F. Kirk.

Foreign Distributors **UK & Europe:** Homecomputer Vertriebs HMBH 282,Flugelstr. 47,4000 Dusseldorf 1, West Germany; Euro Computer Shop, 182 Rue du Faubourg St. Denis, 75010, Paris, France; La Nacelle Bookstore, Procedure D'Abonnement 1-74, 2, Rue Campagne – Premiere, F-75014, Paris, France; Computercollectief, Amstel 312A, 1017 AP Amsterdam, Netherlands. **Asia & Australia:** AXCII Publishing, Inc., 4F Segawa Bldg. 5-2-2, Jingumae, Shibuya-Ku, Tokyo 150, Japan; Computer Services, P.O. Box 13, Clayfield QLD 4011, Australia, Computer Store, P.O. Box 31-261, 22B Milford Rd., Milford, Auckland 9, New Zealand. *(Write for Canadian Distributors)*

EDITORIAL

Changes come and trip all your traps. You retire to write that novel and you find yourself behind a desk again, doing calisthenics with metaphorical braces on your teeth. Life's like that.

It's like this.

This magazine began over eight years ago as *Dr. Dobb's Journal of Tiny BASIC Calisthenics and Orthodontia*, its slogan being "running light without overbyte," and its notion being to promote the kind of programming gymnastics that didn't waste a byte of then-precious memory. It was a good notion, and the best programmers on microcomputers gladly shared their insights on the pages of *Dr. Dobb's* without monetary remuneration. Over the years the magazine continued to distribute, discuss and promote exemplary software: Tiny BASIC made way for Tiny C, a miniature version of Ada and other virtuoso performances in computer calisthenics and orthodontia.

But changes come. Within the past year, *Dr. Dobb's* has started paying its contributors, expanded its editorial staff and become affiliated with M&T Publishing, a profit-making company. It seems fair to ask how much of the original spirit of the publication can survive such changes.

It's a question whose answer I care about. I've been reading and enjoying *Dr. Dobb's* for years, and recently, in the process of researching a book on the history of the microcomputer, I read all the early issues of the magazine. I came to love *Dr. Dobb's Journal* for its irreverence, its personality, its exuberant embracing of tough problems for their very toughness. I don't want it to lose those qualities.

It's not hard to imagine some sad scenarios for *Dr. Dobb's*. It abandons 8-bit software as obsolete and becomes a machine-specific magazine dedicated to some well-known 8088-based personal computer. It ignores any advances in computer architecture since 1979 and lives out its days recycling IMSAI technical notes. It "targets the home [or small business or knowledge worker] market" and disappears in the newstand collage. It fancies itself *Time* magazine and puts red borders on its covers.

Happily, I can promise that none of those scenarios will come to pass in the forseeable future. Who I am is Mike Swaine, former Senior Editor at *InfoWorld*, and as of this issue, Editor-in-Chief of *Dr. Dobb's Journal*. My job is a new one; Renny Wiggins is still Editor. Renny promises, too.

Changes come, and significant changes are ahead for *Dr. Dobb's*, but changes, I think, that respect the magazine's traditions. You can expect the Doctor to experiment with some techniques (like an electronic bulletin board) for getting his methods of software distribution into the 1980's, but don't imagine that he'll abandon his interest in cheap, public-domain software. Expect coverage of the Macintosh computer, not because it's popular, but because good programmers are starting to do amazing things with it. Expect to see some industry leaders in the pages of the magazine, but not at the expense of new writers with something to say.

Expect to see coverage of good implementations of languages and system software and utilities, and occasionally some really unusual things. For example, this month, Renny has put together a rather orthodontic issue, including an introduction to Modulo-2 for Pascal programmers, Ray Duncan's piece on converting to the Forth-83 standard and an unusual article by Richard Grigonis of Children's Television Workshop on what he calls "sixth generation computers."

Next month we'll examine a much-discussed and much-purchased piece of software, Borland International's Turbo Pascal. If half the things that are claimed about it are true . . . but we'll find out next month.

Mike Swaine

This Month's Referees

David Cortesi, Contributing Editor
David Clark, Pennsylvania State University
Clay Phipps, ACM
Henry Socha, SochaLogical Research

Announcing *Dr. Dobb's*

Fifth-Generation Programming Competition

The Challenge: To write a program that applies artificial intelligence techniques to a practical problem, and to extend the present boundaries of what a microcomputer can do.

No rigorous definition of the field of artificial intelligence exists, but there are some programming problem domains that are commonly referred to as AI. These include, among others, pattern recognition, learning, logical reasoning, the representation of knowledge, planning and problem solving. Programs that make a stab at dealing with natural language or with speech or with visual input are regarded as AI programs, as are automatic-programming tools and chess programs.

AI's earliest successes were in game playing, but recent work in the area of expert systems has shown that artificial intelligence techniques can be of real use in practical real-world problem areas like medical diagnosis, geological research and chemical identification. Perhaps because of the success of expert systems, which typically require the processing of large amounts of information, the belief is widespread that microcomputers are inadequate for AI work. We disagree. We believe that there are AI problems waiting to be solved via microcomputer, and we conceive this competition to challenge the brightest microcomputer programmers we know of – *Dr. Dobb's* readers – to find and solve some of those problems. A fantasy? Perhaps. But *Dr. Dobb's Journal* has been publishing Realizable Fantasies since 1976.

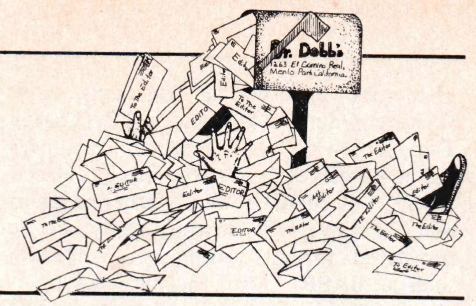
The Rules: The following items must be included in each entry: a functioning program on disk, a well-documented listing of the program, a prose description of the program explaining what it does and how it does it, and the entrant's name and address. The program must be written in a high-level language for a microcomputer, and the entry must be postmarked no later than September 30, 1984 and received by October 30, 1984 (*overseas entrants take note*).

Although your choice of specific language, hardware, problem domain and approach to the problem are all open, we offer some guidelines as to how we will be judging the entries. Your program will be judged for originality and significance of contribution first, and second on its intrinsic merits as a program: modularity of design, efficiency, portability, clarity, and so on. A program that executes in 64K of memory will get higher marks than an equivalent program that requires 96K, but a 128K program that does significantly more may rate higher than either. If your choice of hardware or disk format is an unusual one, you might query us before submitting your entry.

The Payoff: The winner will receive a $1000 prize and will have the program and prose description published in *Dr. Dobb's*. In the tradition of *Dr. Dobb's*, this contribution to the advance of microcomputer software is to be placed in the public domain with full credit to the author. Support for the concept of public domain software is part of *Dr. Dobb's* heritage, but it may not be part of yours. This competition is for those who genuinely want to make a contribution to the advance of microcomputer software. Send your entry to:

Dr. Dobb's Journal, 2464 Embarcadero Way, Palo Alto, CA 94303.

LETTERS



The editorial response card is a great way to talk to us, but don't forget that Dr. Dobb's Journal *also welcomes letters to the editor as a forum for ideas, innovations, irascibility and even idiosyncrasies. Some letters may be edited for clarity and brevity. The Doctor likes hearing from you – keep on writing.*

Ackermann Elaborated

Dear Editor:

I am writing in response to the letter from Mr. Paul Condon (*DDJ* No. 88, February 1984). To logicians, Wilhelm Ackermann is best known in the context of Hilbert and Ackermann, *Mathematical Logic*, 1928. In 1926 David Hilbert asked whether all computable functions are primitive recursive. Ackermann found the answer in 1928, with Ackermann's function. (A translation of Ackermann's original article may be found in van Heijenoort, *From Frege to Gödel*, Harvard University Press, 1967. Ackermann, W., "On Hilbert's Construction of the Real Numbers.")

Primitive recursive functions are, roughly, those that can be computed from the simple functions of sum, product, power, and the operations of substitution, composition, and induction upon these. Ackermann's function is clearly computable, but can be shown to "increase" more rapidly than any primitive recursive function. (A good reference is Hans Hermes, *Enumerabililty, Decidability, Computability*, Springer-Verlag, 1969. Also many texts on formal language theory discuss Ackermann's function.)

The interest in the question of whether or not there exists a recursive, but not primitive recursive, function lay in determining if the mathematical definition of primitive recursion corresponds to our intuitive notion of "function computable by algorithm." Ackermann's function shows that primitive recursion does not; Church's Thesis, almost universally accepted, is the view that the broader class of recursive functions does correspond. There can be no exact mathematical demonstration that some class of mathematically definable functions does correspond to our intuitive ideas of computation, since the latter are necessarily imprecise.

Recursive functions expand the class of primitive recursive functions by allowing for the calculation of the least y such that $f(\bar{x}=y)$ in, for example, the domain of natural numbers. Ackermann's function can fairly easily be seen to be recursive; it can be calculated from the primitive recursive functions with the addition of this minimization feature. What this shows is that the induction in the definition of Ackermann's function is not "ordinary" induction.

Perhaps the minimization feature of recursive functions is what Mr. Condon is getting at with his concluding remark about the correctness or incorrectness of certain definitions of Ackermann's function. I haven't seen the definitions in question.

Beyond the testing of recursion in computing languages that he mentions, the only "use" that I know for the function is the theoretical one of categorizing classes of recursive functions. It is quite likely that Ackermann's function has some wider applications in recursion theory of which I am unaware; I can envisage applications in the study of proof theory for infinitary languages, via Gödel numbering, or just in infinitary proof theory. It sure is fast, isn't it?

Sincerely,
Jay Halcomb
Philosophy Department
University of Arizona
Tucson, AZ 85719

Dear Editor,

In a recent letter, Paul Condon inquired about the origin of Ackermann's function and asked of what use it is. Well, the story goes back to the mid 1920's when Wilhelm Ackermann was working with David Hilbert on an ambitious program to put the foundations of mathematics on a firm axiomatic footing. Among several topics they studied was a class of arithmetic functions, defined on the non-negative integers, which they called "primitive recursive functions."

An exact characterization of the primitive recursive functions is a little too technical for this note. Roughly speaking, however, they are anything you can get by starting with just constants and the successor function $S(n) = n + 1$, followed by rules of composition that allow one to plug one function into another as an argument and to define new functions by induction through the scheme $f(n + 1) = g(n, h(n))$ where g and h are functions previously defined by the same rules.

It turns out that the primitive recursive functions form a very large class. At that time it looked as though any function at all that was defined on the natural numbers and for all values of its arguments might in fact be primitive recursive. This question was raised by Hilbert in 1926. In particular, he asked: Can recursion be used to define a function that is not primitive recursive? The answer was given in 1928 when Ackermann came up with an example of such a function. But the original Ackermann function was not the same as the version that is currently fashionable. The original function had three arguments instead of two. The form currently popular (as quoted by Condon) is apparently due to Rózsa Péter in 1935, although she may have published earlier in some obscure Hungarian journal.

$$\begin{aligned}\mathrm{Ack}(i,j) = {} & \text{if } i = 0 \text{ then } j + 1 \text{ else} \\ & \text{if } j = 0 \text{ then } \mathrm{Ack}(i-1, 1) \text{ else} \\ & \mathrm{Ack}(i-1, \mathrm{Ack}(i, j-1))\end{aligned}$$

Using Péter's form, suppose one defines a new function A of one argument by putting $A(n) = \mathrm{Ack}(n,n)$. Then it can be shown (but not in this note) that the function $A(n)$ grows more rapidly with n than any possible primitive recursive function of one argument, and therefore that A is not itself a primitive recursive function. This is what Ackermann proved and what his function was designed to illustrate.

The growth rate of $A(n)$ is really mind-boggling. For comparison, consider the function $B(n) = n!!!\ldots!!$ where there are n occurrences of the factorial sign. Now $B(n)$ is primitive recursive and its first few values are $B(0) = 0$, $B(1) = 1! = 1$, $B(2) = 2!! = 2$, and $B(3) = 3!!! = \ldots$ well, $B(3)$ has 1747 decimal digits so I won't print it here. So $B(n)$ gets out of the starting gate fast and grows at such a fantastic rate that if the entire universe were filled with high-density floppies they wouldn't be sufficient to store the value of $B(4)$. On the other hand, the value of $A(3)$ is only 61, so $A(n)$ gets off to a slower start. But $A(4)$ is already enormously greater than $B(4)$, and $A(5)$ makes $B(5)$ look like an infinitesimal quantity.

The fact that Ackermann's function is defined by a double recursion makes it useful as one kind of check on correctness of the implementations of recursion in language interpreters and compilers. However, a recursive form of definition, of course, does not mean that a recursive language is necessary to evaluate a function (although it is sometimes not obvious how to do so in a nonrecursive manner). I

offer two little Pascal functions, ack1 (Listing One, below) and ack2 (Listing Two, below), each of which computes the same values as the recursively defined Ack(i,j) discussed above. Neither program uses recursive calls, so they easily can be translated for execution into languages such as BASIC. The function ack1 operates by "faking" an evaluation stack. The function ack2 is more mysterious because it uses so little storage and runs so much faster than the function ack1. The interested reader might want to figure out what is going on in these programs by inserting print statements that show the contents of the arrays after each iteration.

Sincerely,
Milton W. Green
440 Sherwood Way
Menlo Park, CA 94025

Telegovernment?

Dear Editor,

We are in a period where many voters feel their vote means nothing. They do not feel their elected representatives care what their constituents want. As a consequence more and more people have stopped voting.

Over the years the political control of our nation has become ever more centralized in the federal government. At the same time the population has become more decentralized politically. Who is your representative? Does your representative make any effort to keep up to date with the consensus of opinion in his/her district? What methods does your representative use to see that the people he/she represents know what he/she is doing?

Today, we have the basis for alternative methods of participation in government, as well as in other political activities. Computer-based telecommunications combined with television, either public or commercial, can provide the communications links required to return government to the people. An ever increasing number of families now have a computer or terminal in the home.

The time to start taking action regarding the application of telecommunications to government is now. By the time requirements and procedures have been decided we will already have enough users to implement any determinations made. It's also important to prevent, through early action, perversion of the potential power of political telecommunications by power groups.

One method of implementation might be to require every representative to maintain a telecommunications computer as part of his office equipment. This would be accessed from his district through an 800 number, the maintenance of which could be a part of the requirements of doing business, as a local or national monopoly, for the telephone companies.

We have entered a period of technology where government in the United States can once again be of the people, for the people, and by the people. The development of television as an educational medium, teletext, and computer-based telecommunications makes this potentially possible. The continuing reduction in hardware costs makes it economically viable.

This proposal does not advocate an end to representative government at any level. Instead it would return the same control to the electorate enjoyed by the electorate in 1776. We would still need our present legislative, judicial, and executive institutions as they are now constituted; direct democracy allows no check or time delay on temporary and emotional public reaction.

It's necessary for us to implement special education on various policy problems. The need for national "town meetings" has reached a critical level.

It is my wish to start a national organization to investigate, analyze, discuss, and possibly start implementation of public participation in government through telecommunications, television, etc. Anyone interested in participating should feel free to contact me. It will be difficult for me as an individual to provide the initial start of such an organization. This is based on both time and finances available, but I will endeavor to do my best.

Yours truly,
D. R. Crago
P. O. Box 728
Placentia, CA 92670

DDJ

Letters (Text begins on page 9)

Listing One

```
function ack1(i, j: integer): integer;
   var p: integer; a: array[0..4000] of integer;
   begin
      a[0] := i; a[1] := j; p := 1;
      repeat
         if a[p-1] = 0 then begin
            a[p-1] := a[p]+1; p := p-1 end else
            if a[p] = 0 then begin
               a[p] := 1; a[p-1] := a[p-1]-1 end else begin
                  a[p+1] := a[p]-1; a[p] := a[p-1];
                  a[p-1] := a[p-1]-1; p := p+1 end
      until p = 0;
      ack1 := a[0]
   end;
```

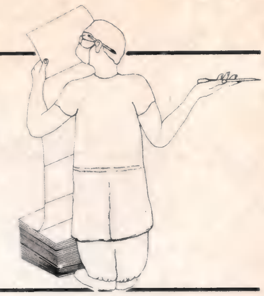
(End Listing One)

Listing Two

```
function ack2(i,j: integer): integer;
   label 1,2,3;
   var p: integer; a,b: array[0..6] of integer;
   begin
         for p := 0 to 6 do begin a[p] := 1; b[p] := -1 end;
      1: b[0] := b[0]+1; a[0] := b[0]+1; p := 0;
      2: if (i=p) and (j=b[p]) then goto 3;
         if b[p] <> a[p+1] then goto 1;
         p := p+1; b[p] := b[p]+1; a[p] := a[0]; goto 2;
      3: ack2 := a[0]
   end;
```

(End Listings)

DR. DOBB'S CLINIC



by D.E. Cortesi, Resident Intern

Optimizing Compilers

We asked for help finding a reference to what we thought was the first optimizing compiler, and we got some good responses, of which the most notable came from C. W. Medlock:

"I believe you are referring to IBM's Fortran H compiler for the IBM 360/370. As to the reference to this work that you are seeking, I believe that you are referring to our paper, "Object Code Optimization," *Communications of the ACM*, Vol. 12, No. 1 (January 1969), written by myself and E. S. Lowry.

"It would be unfair, however, to claim that we were co-authors of the *first* optimizing compiler. Indeed, credit belongs to such early pioneers as John Backus and his original crew for the fine job they did in generating the most-efficient code that the early versions of even IBM 704 Fortran turned out. Many think that it was the quality of this code (compared to that produced by earlier compilers) that convinced the computing community of the feasibility of using high-level languages at all!

"While we know of no other full use of these techniques in another compiler (and quite frankly wish we could find such a compiler for the C language for the IBM PC), there is at least one company, Computer Innovations, that claims to have such a compiler. We haven't been able to test it to see just how 'optimized' the code it produces really is.

"As to the optimization technique that Ed Lowry and I developed, it relies heavily upon tracing the flow of both data and control within a program to see which code items can be moved to a less-frequently-traversed portion of the program (or eliminated entirely as either redundant or producing output that would never be used), and to assist with the temporary assignment of variables to registers over the most frequently executed portions of the code. This analysis was performed over the entire compilation and the technique was thus called 'Global Program Optimization'."

Another reader, Owen Montgomery, also wrote to point out the importance of the early Fortran compilers. He says, "Fortran I was the first algebraic optimizing compiler. At the inception of the Fortran project by John Backus's team at IBM in 1954, most programming was done in machine code or simple assemblers. So-called automatic coding systems existed but would be regarded as little more than symbolic assemblers with built-in function libraries by today's standards. Fortran, to be successful, had to produce very efficient object code. If not optimum and on a par with hand-coded programs, it would surely fail. The consensus among programmers of the day was that efficient programming could not be automated."

Owen's letter included an illuminating quote from a paper by Backus, "The History of FORTRAN I, II, and III," *ACM SIGPLAN Notices*, Vol. 13, No. 8, August 1978. Read it carefully:

"It was an exciting period; when later on we began to get fragments of compiled programs out of the system, we were often astonished at the surprising transformations in the indexing operations and in the arrangement of the computation which the compiler made, changes which made the object program more efficient but which we would not have thought to make as programmers ourselves . . . . Transfers of control appeared which corresponded to no source statements, expressions were radically rearranged, and the same DO statement might produce no instructions in the object program in one context, and in another it would produce many instructions in different places in the program."

A very few times we've had the experience of being delighted by the clever object code a compiler generated, but never on a personal computer. On the contrary, the personal compilers we've used have produced the dullest, most predictable, most flat-footedly literal translation one could imagine.

The entertainment value of the code is not the issue, of course. The crux of the matter is that a genuine optimizer can produce the kind of low-level, foxy cleverness that an assembly programmer sweats to invent – and does it by mechanical methods. That liberates the programmer; the brain power he or she would have spent on achieving tight code can instead be applied where it really matters: on the algorithm.

What we'd like to know is, if Backus could accomplish this in 1954; if Lowry and Medlock could do it in 1969; if the techniques can be found in any of a dozen computer-science textbooks; *why don't we have such compilers today?*

Timeless Language

Meanwhile, we've been exploring the other end of the language spectrum. As part of a continuing (and probably futile) attempt to keep up with the state of the art, your Intern has been studying PROLOG. That's the language that figures heavily in the Japanese plan to produce the Fifth Generation of computers. It is, effectively, an executable variant of the notation of format predicate logic.

At the moment, most implementations of PROLOG are handcrafted prototypes running in university computer centers, primarily in Europe. However, one implementation is a real, commercial product. Known as micro-PROLOG, it's published by Logic Programming Associates Ltd. (10 Burntwood Close, London, England SW18 3JU; $275, shipping included). We recommend it. The manuals are more than adequate; the software works; and the package was delivered across half the world faster than some we've ordered from across the state.

Micro-PROLOG is a fascinating piece of software in itself, entirely aside from the intrinsic fascination of an entirely new language. At its core is a small, fast Z80 interpreter for a list-processing language, one that is strongly reminiscent of LISP, but augmented with pattern-matching primitives. This implements the core of PROLOG, because the core of a PROLOG interpreter is a recursive, backtracking pattern-matcher.

The rest of the system is implemented in PROLOG. This is impressive, because "the rest" includes everything that makes the system usable, including an interactive front end that hides the LISP-like parenthesized notation under a more natural syntax. This gives one confidence in two ways: it demonstrates that PROLOG is a "real" language, since it can be used to write non-trivial software; and it affords proof that the core system is well-tested, since it can execute its own front end.

PROLOG is something else again, it requires the same kind of head-spinning reorientation that accompanies a move from, say, Pascal to Forth, or from COBOL to APL. The most difficult part, at least for this student, is that PROLOG contains no concept of *time*. Every other language we've used contains an implicit notion of movement in time: the computer will do this *and then* it will do that.

In PROLOG, every statement of a program is a candidate for execution at every moment. Actually, "statement" is a poor term for a programmer to use in describing a line of PROLOG; it carries too many implications from other lan-

guages. "Assertion" is a better term; a PROLOG program is composed of a number of assertions about the world. In principle, all assertions are executed at once, in parallel, and those that turn out to be false are discarded. (In practice, the interpreter tries them sequentially, but it isn't cricket to assume that.) It's this inherent parallelism that makes PROLOG a candidate for the system language of the Fifth Generation machines, but boy! does it twist your mind at first.

The micro-PROLOG system includes a hands-on tutorial that serves to get you just barely started. Two other books may help. *Programming in PROLOG* by W. F. Clocksin and C. S. Mellish (Springer-Verlag, 1981) is a reasonably good textbook on the use of the language. *Logic for Problem Solving* by Robert Kowalski (North-Holland/Elsevier, 1979) is an exhaustive, scholarly exploration of the relationship between Horn-clause logic (which PROLOG implements) and everything else in the world. It is heavy going, and parts are interesting only to an implementor of PROLOG, not to a user of it. Still, parts of Kowalski's book explain the rationale behind this initially puzzling language.

One warning: the syntax of PROLOG is nowhere near standardized. Micro-PROLOG seems to be full PROLOG as described by Clocksin and Mellish (it may even be a superset). However, the syntax of micro-PROLOG has many differences of detail from that used by Clocksin and Mellish, and that in turn is different from the syntax that Kowalski uses. It is not difficult to translate from one to another, but it imposes an extra mental load.

Are any readers also exploring PROLOG? Somebody must be, because our local bookstore can't keep *Programming in PROLOG* in stock. Write and share your experiences.

Prissy Program?

Oscar Goldman writes to ask, "Did you know that RMAC is a delicate flower who turns her (his? its?) head in embarrassment at the presence of an error? Take a look at this listing:

```
0000 3A0600 lda  store
     inra   ; typographical error
0003 320600 sta  store
0006 00     store: db 0
     end
```

It took me quite a while to debug a program which had just that typo (of 'inra' for 'inr a'). I think you might warn your readers of this one."

With pleasure, Oscar. What we have here is the result of the peculiarly flexible syntax of the Digital Research assemblers.

Typical assemblers require a label to begin in column 1 of a statement. That permits a simple test for the presence of a label: if column 1 is blank, there isn't one. The DRI assemblers (ASM, MAC, RMAC) have a more complicated rule. They isolate the first word of a statement and look for it in the table of known operation codes and in the table of defined macro names. If the first word is not an opcode or a macro, it must be a label.

That's what happened here: since "inra" isn't an opcode, RMAC decided it must be a label. It shows up in the symbol table listing. The fact that it was indented to line up with the other opcodes made no difference at all.

This syntactic scheme can be useful; you can define pseudo data structures, indenting inner names under the names that contain them. However, as Oscar notes, it can also let you create some very tricky bugs.

I Tell You Three Times

Another correspondent, Michael Barr, writes with a tale of keyboard handling in MSDOS 2.0. We don't have an MSDOS machine on which we can test his claims, but perhaps a reader can test and explain them. Here is his account.

"I have discovered a peculiar effect of control-S. To see it, at the system prompt simply type control-S. Do it again, then a third time. Try it three more times. Now press control-S, followed by any ordinary character, then control-S again. Confused? So was I.

"There are two DOS functions that are relevant. The first is function 7, which goes into a loop until you type a character and then returns that character. It works as documented. The second is function

11, which is documented to return a true flag only if there is a character waiting in the keyboard buffer. What it actually does is another matter.

"After the first control-S, that character is in the buffer (as can be determined with either function 6 or function 7), but function 11 swears the buffer is empty. After another keypress, whether control-S or another character, assuming you have not gotten the first control-S out of the buffer, the buffer is *empty* while function 11 returns a *true* flag.

"What happens after the third keystroke? Well, that is exactly like the first. Why, then, do you actually get a character that third time? I can only speculate that the command interpreter in DOS operates in the same way as a program of mine. What my program did was to call function 11 until returned a true flag, then call function 7 to get the character. That way the program could do something else instead of just idling until there was input. (It wasn't doing anything terribly urgent – it was polling location 0000:0417 hex to find out if the Caps Lock key had been pressed.)

"The first entry of control-S had no effect on function 11, while the second caused function 11 to return a true flag with the buffer empty. Then the program called function 7, which just idled until another character was entered. Thus every third keypress was captured. I speculate that the DOS command interpreter is doing likewise."

We can't check out Barr's problem, but here's a hint for any reader who wants to look into it. It's suggestive that the problem centers on control-S. That's XOFF, or DC3, and in many systems it would be a signal to freeze screen output until a control-Q (XON, DC1) was entered. MSDOS contains a number of half-assimilated CP/M compatibilities (Barr also mentions the undocumented feature that control-P will toggle printer echo of the screen output, which is a CP/M convention). CP/M 2 uses control-S to freeze the screen, releasing the screen when any further key is hit. Both keystrokes are swallowed in the system. Maybe that's what MSDOS is trying to do, but not thoroughly enough.

Computer Citizenship

However, another point in Barr's letter raises our hackles. In our oh-so-humble opinion, polling loops are bad! A program that refuses to read the keyboard until its poll shows that a character is waiting is a bad citizen; it defeats any number of nice features that the DOS could otherwise implement. It defeats the largest benefits of a buffered keyboard (see this column for May 1983, under the heading "Software vs. Common Sense").

Furthermore, a polling program is absolutely deadly in a multitasking system. It always uses its full slice of time even when it has nothing to do; that defeats the basic idea of multitasking, which is that one task can make good use of another task's idle time. And multitasking systems are coming to micros: Concurrent CP/M is available now, and MSDOS 3.0 is rumored to be one.

Sure, a Caps Lock flag on the status line is a cute feature, but consider: Under Concurrent CP/M, a program like Barr's would be checking the Caps Lock byte at times when the user was entering data to a completely different program. The same problem would probably occur under Microsoft Windows(tm) or VisiOn(tm) – it would be polling Caps Lock when the cursor was in another program's window.

This is an example of how a solution can go wrong when it is implemented at the wrong level of the system. The PC keyboard lacks two 29-cent LEDs to display the state of Caps Lock and Num Lock. The PC operating system does nothing about it. So application programmers try to make up for it, either with a polling loop or by stealing the keyboard interrupt vector. And all the ones whose efforts succeed end up with programs that flat *won't work* in systems where the hardware is a resource that has to be shared among multiple programs. DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 190.**

CP/M EXCHANGE

by Robert Blum

A few months ago, Terje Bolstad of Norway wrote to ask if I had any experience with an undocumented device he had found in PIP called IRD:. Unfortunately, I had no prior knowledge of the IRD: device to share but was intrigued, as many of you have been, with its origin and possible purpose. I decided to go directly to the mountain for further advisement.

My first letter to DRI was promptly answered. Rather than being provided with an answer, however, I was referred for an explanation to the OEM from whom I had purchased my CP/M system. Since several of us had already done considerable research on how IRD: worked, I didn't feel that a side trip to another vendor was necessary. I wrote to DRI again to explain more fully what steps had been taken to decipher the mystery device's purpose. The response to my second letter came via a phone call, but to no avail because the answer was considered to be proprietary information of DRI.

One further encounter with DRI proved to be no more fruitful than the first two. Undaunted, I took my quest for knowledge to the readers of this column in January. To stimulate the juices of adventure I offered a *DDJ* tee shirt to the author of the best answer.

I should have known what would happen next. However, I was caught off guard when my mail box began to fill with more than my usual quota of correspondence. Your responses to my request for help ranged from a few lines jotted on a postcard to catalog-sized envelopes overstuffed with program listings. As is always the case, and very pleasantly so, when dealing with *DDJ* readers the answers were most insightful and gave evidence to a great deal of work.

It would be impossible to print all the responses that I received. With the exception of some editing, what follows is a cross section of your answers to the IRD: mystery.

IRD: No Longer A Mystery

Dear Bob,

I've just seen your column in the January issue of *Dr. Dobb's*. There really is no mystery about the device called IRD:. The letters "IRD" stand for "Infra-Red Detector." The IRD: device represents a peripheral device whose purpose, clearly, is to monitor the infra-red level in its vicinity. This can be used for various purposes, for instance, to shut down the computer when the programmer's temperature rises above a certain critical level because of CP/M's idiosyncrasies. But the truth, actually, is this: In the beginnings of CP/M, Kildall and associates used to engage in target practice using guns which emitted infra-red (instead of water) and had several IRD's installed in the targets to record their hits.

Oscar Goldman

Dear Robert,

I was so interested in your PIP mystery device that I had to solve the problem before I went to sleep. It seems that the IRD: driver is for the Intellec-8/Icom Reader. On my system PIP hangs (actually it keeps returning 07fh, which never signifies end of file). A very tight routine within PIP is entered and then sometimes never exited when the IRD: device is used. See Listing One (page 18) for the IRD: subroutine taken from PIP version 1.5.

Craig D. Miller

Dear Robert,

Internal Research and Development? Infra Red Device? Irreversible Route to Delirium?

No, the mystery device is, in fact, the Intel Paper Tape ReaDer. Does anyone remember paper tape? Yes, PIP contains a built-in hardware driver for this device which is (was) used with the Intel Intellec-8 MDS system. At least this was true with the V1.4 version of PIP, and I assume the CP/M 2.2 version follows suit.

As for the details, the following actions are performed when each byte is read from the IRD: device during a PIP transfer:

(1) The value OCH is output to port 01H.

(2) The value 08H is output to port 01H. This strobes the paper tape reader mechanism.

(3) PIP loops until bit 5 of port 01H is set to 1. This is the Ready bit.

(4) Finally, the lower 7 bits of port 03H are returned as the input value.

Note that if a system does not contain I/O ports 1 or 3, the net effect of all this will be to return a neverending string of 7fh's, which is what you observed.

IRD: is, no doubt, useless on 99% of CP/M-based systems. In fact, it could be dangerous on systems which use these I/O ports for other purposes.

Now that you know about the Irresistible Riddle to Decode device, you might want to buy an Intellec-8 system in order to realize the full potential of your CP/M software.

Rick Hollinbeck

Dear Mr. Blum:

I think I can shed a little light on the mystery of the IRD: device in CP/M 2.2's PIP.COM. The code appears to drive a strobed parallel input port device. To the best of my recollection, the earlier Intel MDS/Intellec systems have a paper type reader addressed at these port addresses. This suggests that IRD: stands for Intel ReaDer. The reason it performs differently on various systems now becomes clear – it depends on what sort of device (if any) you have addressed at these port addresses. If you don't have any device on port 03h, you will (hopefully) input an FFh which will be masked to 7Fh, which is what you got in your disk file.

This is most likely a remnant of the days when CP/M was still being developed, and that is why Digital Research doesn't like to discuss it. Remember, the original CP/M was primarily written in PL/M, hosted on an Intel development system. They had to read the CP/M utility object tapes somehow, and this is probably how they did it.

Terence M. Kennedy

Dear Robert,

It should be obvious to everyone that IRD: is DRI spelled backwards. However, since acronyms have to stand for something, and Inc. Research Digital sounds really horrible, I propose we christen our device (DRI seems to want to disown the poor child) "Interminable Repetitious Device," in honor of its propensity for generating an endless stream of s's.

I'm sure that if we devote enough collective effort, we can find practical uses for IRD:. In fact, I have already thought of one: By PIPing IRD: to a disk file, we have a convenient, fast way of generating a large file in order to test some program.

David Kettle

Dear Bob,

I suspect that IRD: is simply DRI backwards. If it has to have a meaning, perhaps it would be "Incorporated Research Digitalizer," or some such.

Roy Lipscomb

Dear Sir:

ARD:, IRD:, PRN: – three popular peripheral devices.

This is explained in the documentation for CP/M V1.3, copyrighted by Digi-

tal Research, dated 1976. When the new guide was printed and stapled into a neat booklet the meanings were deleted and PIP.COM was improved. Apparently, PIP's internal table was not completely changed, thus a remnant still exists:

ARD: Addmaster paper tape reader

IRD: Intel or Icom paper tape reader (these two were considered high-speed vs. the ASR-33)

PRN: Tally hard copy printer

There is also a table of error messages in PIP.COM that seem peculiar because I have not seen them reported before: tape stopped or illegal HEX character encountered. The document also explained this in the paragraph about PIP.COM special functions when processing a punched paper tape. I now suspect they are related to ARD: and IRD: as they were never used when I assigned a cassette tape to PUN:/RDR:.

As you may have guessed, I have been an owner of a CP/M operating IMSAI system since November 1976 and, though not having an ASR-33 as a terminal, I did experience punched paper in those historical days.

Delmar Rawson

There you have it, Terje. IRD: stands for Intel-Icom-Intellec-8 ReaDer and was used on that system for reading paper tape. Why all the secrecy about the IRD: device? All we can do is guess, but I am betting that DRI still has an Intel system or two being used daily by one of Jerry Pournelle's mad friends.

Before I forget, my heartiest congratulations go out to Delmar Rawson. He was not only fast but also correct. His *DDJ* tee shirt is on its way.

I would also like to express my thanks to each of you who helped me in finding the answer to the mystery IRD: device.

CP/M Tidbits

Included with Terence Kennedy's response to the IRD: mystery were a few insights into how CP/M does its job:

"In reference to another mysterious topic of discussion in recent months, the reason some programs appear to 'eat' one character before warm booting is because the BDOS itself is reloaded. Let me digress for a moment.

"In CP/M 2.2, there are three sanctioned ways to perform console I/O: via 'normal' BDOS console I/O calls, via BDOS function number 6, and by calling the BIOS directly. In fact, the last two methods do exactly the same thing. I presume Digital Research was preparing us for systems like MP/M and CP/M 3, where accessing the BIOS is a definite 'no-no.'

"Normal BDOS I/O works as follows: whenever a character is sent out to the console, a check is made for an input character being available; it is tested to see if it is a Control-S. Note that this is the only character that the BDOS is looking for at this time. If it is a Control-S, the BDOS enters a tight loop looking for the next character, which may be, for example, a Control-Q to resume output or a Control-C to warm boot. However, if the input character was not a Control-S, it is stored in a one-byte buffer in the BDOS, and the BDOS no longer performs any status checks during output. When the .COM program terminates with a RET to the CCP (as in STAT), the character is still in the BDOS's buffer, and the next BDOS input call will retrieve it first, before calling the BIOS for more characters. If, however, the .COM program terminates with a warm boot, the BDOS is re-loaded, leaving the buffer empty; hence, one character is gone.

"This also causes an unfortunate interaction between BDOS function 6 I/O and the rest of the BDOS I/O calls. Programs should never mix 'normal' BDOS calls with function 6 or BIOS calls. Otherwise, input characters may 'vanish' mysteriously, only to reappear at a later input prompt.

"Digital Research has a stop-gap patch for the latter problem, but only for MP/M II 2.1. They have apparently taken a completely different approach in CP/M 3.

In 3, they appear to examine characters as they do in 2.2; but if it is not an appropriate control character, it is discarded. This is why type-ahead routines which worked with 2.2 will not function at the CCP level in CP/M 3.

"Here are two other tidbits that may be of interest to you.

"The CP/M 2.2 'Reset Disk System' BDOS function 13 returns a result parameter in the A register. A is 0 if no $$$.SUB file is active; otherwise it is nonzero. This is how the CCP knows whether or not to continue from the submit file. However, the function only looks at whether or not the first character of any filename is a dollar sign. This is why you may have noticed some strange extra disk activity and a general sluggishness if you have, for example, a filename beginning with a '$' in your directory. It appears that the CCP attempts to open $$$.SUB, fails, and then attempts to delete $$$.SUB.

"The CP/M 2.2 'Raw Console I/O' BDOS function 6 has an undocumented additional option. An E register argument of OFEh merely checks the status of the console, and does not return or delete any pending input characters. This is consistent with raw character handling in CP/M 3 and MP/M II."

OUT: and INP: – Making Them Work

Craig Miller also found the time to include this brief explanation and the necessary assembler source code to implement the OUT: and INP: devices in PIP.COM:

"When DRI upgraded PIP for CP/M Plus, the undocumented IRD: device was dropped and is now only available in V2.2 of CP/M. I have found that the OUT: and INP: drivers are in PIP 3.0; however, the documentation doesn't describe how to install them. The enclosed listing (Listing Two, below) shows how to do this."

DDJ

CP/M Exchange (Text begins on page 16)

Listing One

```
         MVI     A,0CH           ;Turn reader on
         OUT     01H
         MVI     A,08H           :Turn reader off
         OUT     01H
WAIT:    IN      01H             ;Wait until reader has a character
         RLC
         RLC
         RLC
         RAR
         JC      INPUT           ;If carry, then done (get char), else loop
         JMP     WAIT            ;Loop back
INPUT:   IN      03H             ;Get character
         ANI     07FH            ;Mask parity bit
         RET
```

(End Listing One)

Listing Two

```
;THIS FILE WILL PATCH THE I/O ROUTINES INTO
;PIP FOR VERSIONS 2.2 AND 3.0 (INP: AND OUT:).

TRUE            EQU     0FFFFH
FALSE           EQU     0

PIP22           EQU     FALSE
PIP30           EQU     TRUE

                IF      PIP22
PATCH           EQU     110H
PATCHEND        EQU     1FFH
INPJMP          EQU     103H
INPCHR          EQU     109H
OUTJMP          EQU     106H
                ENDIF

                IF      CPM30
PATCH           EQU     188H
PATCHEND        EQU     1FFH
INPJMP          EQU     180H
OUTJMP          EQU     184H
                ENDIF

                ORG     INPJMP
                JMP     INPUT

                ORG     OUTJMP
                JMP     OUTPUT

                ORG     PATCH

DATA            EQU     84H
STAT            EQU     85H
MODE            EQU     86H
```

(Continued on page 20)

CP/M Exchange (Listing Continued, text begins on page 16)

Listing Two

```
COMM        EQU     87H

INIT:       MVI     A,0CEH          ;INIT 2661 TO 9600 BAUD, 8 BITS
            STA     INITFLG
            OUT     MODE
            MVI     A,03H
            OUT     MODE
            MVI     A,27H
            OUT     COMM
            IN      DATA
            IN      DATA
            MVI     C,9
            LXI     D,SIGNON        ;PRINT THE PATCHED MESSAGE
            JMP     5

INPUT:      LDA     INITFLG         ;INPUT A CHARACTER FROM DPC-100
            ORA     A
            CZ      INIT
INPLOP:     IN      STAT
            CMA
            ANI     3
            JNZ     INPLOP
            IN      DATA
            ANI     7FH

            IF      PIP22
            STA     INPCHR
            ENDIF
            RET

OUTPUT:     MOV     A,C             ;OUTPUT A CHARACTER
            STA     SAVCHR
            LDA     INITFLG
            ORA     A
            CZ      INIT
OUTLOP:     IN      STAT
            ANI     1
            JZ      OUTLOP
            LDA     SAVCHR
            OUT     DATA
            RET

SIGNON:     DB      '2661 SERIAL PORT AT 9600 BAUD',0DH,0AH,'$'
INITFLG:    DB      0
SAVCHR:     DB      0

LAST:
            IF      LAST > PATCHEND
            ERROR PATCH IS TOO LONG !!!!!!!
            ENDIF

            END
```

(End Listings)

SUBVERSIVE SOFTWARE

So cheap and useful it's...dangerous!

A radical idea in software development.
Useful Pascal programs, with:

- Complete annotated *source listings;*
- Disk(s) with *source code and compiled code;*
- User manual;
- Complete programmer documentation describing data structures and algorithms; and giving suggestions for modification.

Modify them, include them in your own systems, or simply use them.

A growing library of software tools you'll find hard to resist.

SUBVERSIVE SOFTWARE

TPL

The Text Processing Language. A text-file runoff program consisting of a set of text-processing primitive commands from which more complex commands (macros) can be built (as in Logo). Features include:

- Complete customization of text processing through macro definition and expansion, looping structures, and conditional statements;
- Adapts to any printer;
- Pagination;
- Text justification and centering;
- Indexing and tables of contents;
- Superscripts and subscripts;
- Bolding and underlining;
- Multiple headers and footers;
- End notes and footnotes;
- Widow and orphan suppression;
- Floating tables and 'keeps.'

$50

SUBVERSIVE SOFTWARE

DBX

Blocked Keyed Data Access Module. Maintains disk files of keyed data. Can be used for bibliographies, glossaries, multi-key data base construction, and many other applications.

- Variable-length keys;
- Variable-length data;
- Sequential access and rapid keyed access;
- Single disk access per operation (store, find, delete) in most cases;
- Multiple files;
- Dynamic memory allocation for RAM-resident index and current "page" of entries;
- Includes demonstration program and testbed program.

$50

SUBVERSIVE SOFTWARE

PDMS

The Pascal Data Management System. A user-oriented data management system in which numeric and alphanumeric data are stored in tables with named columns and numbered rows. Currently being used for dozens of different kinds of business and scientific applications, from inventory management to laboratory data analysis. Includes over 20 Pascal programs; more than 10,000 lines of code. Main features include:

- Maximum of 32,767 rows per file;
- Maximum of 400 characters per row, and 40 columns per table;
- Full-screen editing of rows and columns, with scrolling, windowing, global search/replace, and other editing features;
- Sorting, copying, merging, and reducing routines;
- Mailing label program;
- Reporting program generates reports with control breaks, totals and subtotals, and selects rows by field value; many other reporting features;
- Cross-tabulation, correlations, and multiple regression;
- Video-display-handling module;
- Disk-file-handling module.

Many other features. UCSD formats only.

$250

SUBVERSIVE SOFTWARE

ZED

Full-screen text editor; designed to be used either with TPL or by itself.

- Full cursor control;
- Insert mode with word wrap;
- 'Paint' mode;
- Single-keystroke or dual-keystroke commands;
- Command synonyms;
- Global search and replace;
- Block move, block copy, and block delete.

$50

BVERSIVE FTWARE

SCINTILLA

A log logit curve fitting program for radio-immunologic data; must be used with PDMS (described above).

- Multiple protocol files;
- Quality control files;
- Four-parameter non-linear curve fit.

UCSD formats only. **$250**

SUBVERSIVE SOFTWARE

CHROME

Chromatography data analysis program:

- Graphic display of analog data;
- Panning and zooming;
- Automatic peak-finding and baseline calculation;
- Full interactive peak editing;
- Computation of peak areas;
- Strip charts on C. Itoh and EPSON printers.

$100

SUBVERSIVE SOFTWARE

PLANE

Planimetry program:

- Bit-pad entry of cross sections;
- Real-time turtlegraphics display;
- Calculation of areas;
- Saves calculations to text file.

$100

VERSIVE TWARE

MINT

A terminal emulation program for communication between computers of any size.

- User-configurable uploading and downloading of files;
- X-ON/X-OFF and EOB/ACK protocols;
- Interrupt-driven serial input (for Prometheus Versacard in Apple II);
- Printer-logging.

$50

For more information, call 919-942-1411. To order, use form below or call our toll-free number: 1-800-XPASCAL

Check appropriate boxes:

FORMAT

- ☐ 8″ UCSD SSSD
- ☐ 5¼″ Apple Pascal
- ☐ 5¼″ UCSD IBM PC 320k
- ☐ 8″ CP/M SSSD
- ☐ 5¼″ IBM MS-DOS
- ☐ 5¼″ CP/M Osborne

PRODUCT	PRICE
☐ DBX	$ 50
☐ PDMS	$250
☐ TPL	$ 50
☐ ZED	$ 50
☐ MINT	$ 50
☐ SCINTILLA	$250
☐ CHROME	$100
☐ PLANE	$100

Name

Address

☐ MasterCard ☐ VISA ☐ Check ☐ C.O.D.

(Please include card # and expiration date)

SUBVERSIVE SOFTWARE A division of Pascal & Associates, 135 East Rosemary St., Chapel Hill, NC 27514

Circle no. 49 on reader service card.

An Introduction to Modula 2 for Pascal Programmers

If you've been following discussions about programming languages in the computer press, you have probably seen mention recently of "Modula-2, the new language by Niklaus Wirth, the father of Pascal." Articles have appeared that pit Modula-2 against Pascal and even against Ada!

What exactly is Modula-2, and how much does it differ from Pascal? How difficult would it be for an experienced Pascal programmer to learn Modula-2?

We feel that a programmer familiar with Pascal should be able to pick up the basics of Modula-2 in an afternoon and be programming comfortably within a few days. To this end, we'll briefly introduce Modula-2 from a Pascal programmer's point of view, describing similarities and differences as we go.

This will not be an introduction to the entire Modula-2 language. Parts of the language deal with direct access to the hardware and with concurrent programming. These are worth an article by themselves. The definitive description of Modula-2 may be found in Niklaus Wirth's *Programming in Modula-2*, 2nd edition (Springer-Verlag, 1983).

One warning: The following assumes some familiarity with Pascal. In particular, we will refer to what is known as Standard Pascal or ISO Pascal, which is the language standardized by the International Standards Organization (ISO), or the equivalent language standardized in the United States as ANSI/IEEE770X3. 97–1983. Most, if not all, Pascal implementations provide (incompatible) features beyond this standard, but few fail to provide the features we will mention.

The Origins of Pascal

When you hear the phrase "structured programming," chances are good that you'll think of Pascal, although you might think of PL/I, C, or even ALGOL 60 or ALGOL 68. Pascal, designed by Niklaus Wirth, appeared in the literature in about 1971. It traces its structure back to ALGOL 60, although Wirth worked on several languages between 1960 and 1971, including ALGOL-W, EULER, and PL360.

Wirth was dissatisfied with ALGOL 68 (the new improved ALGOL), at least partly because it was incomprehensible even to its designers and it did not admit of a simple, efficient implementation on typical computers. PL/I provided a similar lesson in the costs of large languages. Pascal on the other hand, was designed with simplicity and regularity as primary goals. It is characterized by features whose implementation is well understood and efficient and whose interactions are known and manageable. It is also widely associated with the now famous (or infamous) term "strongly typed language," although it was not the first language of this kind.

The Origins of Modula-2

The roots of Modula-2 are to be found in Pascal, Modula, and Mesa. Modula (i.e., Modula-1) was an intermediate stage in the evolution to Modula-2. Mesa, a language developed at the Xerox Palo Alto Research Center (PARC), was used to program the Alto and Star computers.

In 1976, Wirth spent a sabbatical year at Xerox PARC. He was impressed by Mesa and by the fact that an entire personal computer environment (operating system, compilers, editors, etc.) could be developed using a single high-level language. He returned to the Eidgenoessische Technische Hochschule (ETH) in Switzerland and designed the Lilith machine, a microcoded computer with a high-resolution bitmap display similar to the Xerox Alto. The entire software environment of the Lilith was implemented in Modula-2.

Like Pascal, Modula-2 is a relatively simple language. It contains a few simple logical control structures. It provides a number of data types, including enumerations, sets, and records, and it allows the programmer to define types. In contrast to standard Pascal, it includes a number of features that make it particularly well suited for implementing large software systems, concurrent (multiprocess or interrupt-driven) systems, and systems that require low-level access to the hardware.

Unlike Pascal, the language Modula-2 does not define a file data type or any I/O operations, such as Get, Put, Read, or Write. In a Modula-2 system, **standard modules** provide access to files and devices; these services may even be user-written. The book *Programming in Modula-2* defines several of these standard modules. This approach is familiar to C programmers, who use a standard library to do I/O.

Because of the great effect this has on Modula-2 program portability, several companies that sell Modula-2 implementations are working together to ensure compatibility of both their language implementations and their libraries.

Modula-2 and Modularity

Many dialects of Pascal allow for what is called independent compilation. The parts of a program are compiled independently and then combined with a linker or binder. While full checking is done within each compiled unit, little or no checking is done between units. For example, the number and types of procedure parameters are not checked in a call between units; in fact, it may be possible to use a variable in one unit as a procedure in another, usually with undesirable results!

The Modula-2 language stipulates that a program may consist of compilation units (modules) that are separately *but not independently* compiled. A correct implementation of the language provides as much checking between two separately compiled modules as it provides within each module. A few Pascal implementations (notably UCSD) also provide strong checking between compiled units.

When a module needs to use objects provided by another module, such as types, constants, variables, or procedures, it **imports** them from the providing module, giving the name of the providing module and the names of the desired objects. Such a module is said to be a "client" of the providing module.

Modules that provide objects for use by other modules are divided into a **definition part** and an **implementation part.** The definition part lists the objects that are provided by the module and gives enough information about them to allow the compiler to check their use. The implementation part gives the full story, including procedure bodies and other objects that are hidden within the module.

One important thing to note is that many implementations are possible for a given definition part. The details of an implementation are not visible to other modules, so any implementation that provides the objects described in the definition part and operates in the "expected" way will do.

This is called **information hiding** and it is a very powerful way of structuring systems. Some of its power comes from

by Hugh McLarty
and David W. Smith

Hugh McLarty, Logitech, 165 University Avenue, Palo Alto, CA 94301.

David W. Smith, Tolerant Systems, Inc. 81 E. Daggett Dr., San Jose, CA 95134.

the fact that most bugs, and much of what might be called software inertia, come from assumptions (i.e., information). A bug is the symptom of an incorrect assumption; inertia is the result of a widely embedded assumption that needs to be changed. Information hiding is simply the technique of localizing information; it operates on the theory that information the programmer does not have access to cannot be abused or embedded in the code. Modula-2 modules provide an excellent mechanism for doing this.

For example, during development, simple but logically correct implementations are used. As module interfaces become settled and the needs of the system become understood, more sophisticated implementations can be substituted. Altering or replacing an implementation part does not require that we understand how the rest of the system depends upon it, only that we understand how the rest of the system depends upon the objects in the definition part. If the definition part does not change, we have some confidence that no ripple effects will spread changes (and bugs) across the system.

Consider a symbol table module, as used in a compiler to collect identifiers and their meanings. We can choose an interface and write it as a definition part then create a simple implementation part, perhaps using a linear list to store symbol entries. This is slow for more than a few symbols, but it is quick and easy to implement. Later we can substitute an implementation that uses a hash table or a binary tree, or one that swaps entries to disk when memory fills up, and so on. A well-conceived definition part will remain constant through all of this, and few if any changes to client modules will be necessary.

From Pascal to Modula-2

To begin our discussion of the differences between Modula-2 and Pascal, let's consider the short programs in Figures 1a and 1b (at right). The single-entity PROGRAM of Pascal is replaced by the MODULE of Modula-2. This module is an example of a program module. Other module types are discussed below.

First, upper and lower case are distinguished in Modula-2; in particular, keywords are all upper case. "WriteLn" is distinct from "writeln" and both are distinct from "WRITELN."

The Modula-2 program imports two objects from the InOut module; in this case the objects are procedures. When this program is compiled, the definition part for InOut will be found and used to check the calls of those procedures within this client module. InOut is a standard module, described in the book *Programming in Modula-2*.

This explicit import takes a little more effort to write but quickly tells the reader what objects are being imported and where they come from. The only identifiers that are implicitly imported are record fields and enumeration constants.

Identifiers, Numbers, and Strings

The Pascal Standard says that identifiers may be of any length and that all characters are significant. The original (Jensen and Wirth) definition only required that the first eight characters be significant, and many systems still have a noticeable restriction, especially for identifiers known between modules. The Modula-2 definition does not say anything about identifier length, but it seems to be widely understood that identifiers are very long, i.e., 72 or 80 characters at a minimum. All characters are significant, and a too-long identifier is an error. We have already mentioned the new case distinctions.

Real numbers are just slightly different. No digits are required after the decimal point, but the decimal point is always required. For example:

```
PROGRAM TestProgram (OUTPUT);
BEGIN
    WriteLn('Hello, World');
END.
```

Figure 1a.
A simple Pascal program

```
MODULE TestProgram;
    FROM InOut IMPORT WriteString. WriteLn;
BEGIN
    WriteString("Hello, World");
    WriteLn
END TestProgram.
```

Figure 1b.
The same program in Modula-2

```
TYPE DIRECTIONS = SET OF (up, down, left, right);
VAR barriers: DIRECTIONS;
. . .
barriers := DIRECTIONS {up, left}
```

Figure 2.
Modula-2 set expression

```
RECORD x, y: REAL;
    CASE tag: Color OF                          (* first variant part   *)
      red:          a:   CARDINAL               (* note, no ( . . )     *)
    | blue:         b:   INTEGER
    | green:        c:   BITSET
    END; (* CASE *)
    z: [0 . . 99];
    CASE BOOLEAN OF                             (* second variant part*)
      TRUE:         m, n: CARDINAL
    | FALSE:        p:   POINTER TO CHAR
    END  (* CASE *)
END; (* RECORD *)
```

Figure 3.
Modula-2 record expression

5. is OK in Modula-2 but not in Pascal.

1E-3 is OK in Pascal but not in Modula-2.

More importantly, Modula-2 provides syntax for hexadecimal and octal constants and for writing character constants using their octal ordinal values. Some examples:

hexadecimal:	0FFFFH	
octal:	0177564B	
character:	15C	(ASCII CR)

A character string in Modula-2 is a sequence of characters surrounded by either single or double quotation marks. A string delimited by single quotes may contain double quotes, and vice versa, but there is no way to have both in the same string.

A string of length n (n > 1) has the type:

ARRAY [0 . . n−1] OF CHAR

Presently, a string of length 1 will have type CHAR. This causes some problems, since it is not compatible with any ARRAY OF CHAR type. Wirth has agreed to relax the compatibility rules in this case, and this change should be in the next edition of *Programming in Modula-2*.

Types and Declarations

One radical change from Pascal to Modula-2 is that Modula-2 allows any number of CONST, TYPE, VAR, and PROCEDURE sections, in any order; it is also legal to make forward references within statements (e.g., to types, constants, variables, and other procedures). There is no FORWARD construct in Modula-2.

An important new scalar type, CARDINAL, takes on nonnegative integer values up to an implementation-defined maximum, usually called MaxCard. This is intended to allow use of the unsigned arithmetic that is available on many machines, and it is expected that MaxCard > MaxInt in many implementations. Many things that involve INTEGER in Pascal are done with CARDINAL in Modula-2.

Enumerations are unchanged, but subrange types are now surrounded by [and].

Set types are unchanged, although set expressions are slightly different. A standard set type, BITSET, is defined as

TYPE BITSET = SET OF [0. . n−1]

where n is the size, in bits, of a handy unit on the target computer – typically one or two words. (BITSET is part of the language, but there is no standard constant related to n!) As for set expressions, they can contain only constant elements and ranges and must be preceded by the name of a set type unless that type is BITSET (Figure 2, page 23).

Records have a slightly different syntax and are allowed to have any number of variant parts in any position (Figure 3, page 23). Modula-2 does not support "packed" structures.

Procedures

There are two minor differences between a Pascal procedure and a Modula-2 procedure. The first is that an empty formal parameter list is allowed and, indeed, is required for a function procedure with no parameters. The second is that the name of the procedure must be repeated after the terminating END of the procedure body.

However, a revolutionary change has occurred in the way procedures can be used: In addition to the procedure parameters allowed in Pascal, Modula-2 allows variables and record fields of a procedure type. Such variables and fields can have a procedure assigned to them; the procedure can then be called by designating the variable or field, followed by an actual parameter list. We have suggested some of the possibilities of this change with the fragment of Modula-2 code in Figure 4 (at left).

In a related area, Modula-2 also addresses some of the old (Level 0) Pascal problems with array parameters. In Modula-2, a procedure such as the one in Figure 5 (page 26) is possible. The ARRAY OF construction means that, when the procedure is called, an array of any size may be passed to this parameter. Within the procedure, the array has a lower bound of 0, and the built-in function HIGH returns the appropriate upper bound.

Expressions

In dealing with Modula-2 expressions, the Pascal programmer will find few surprises. The same basic set of operators is available in Modula-2, including the set operators, with the same priorities.

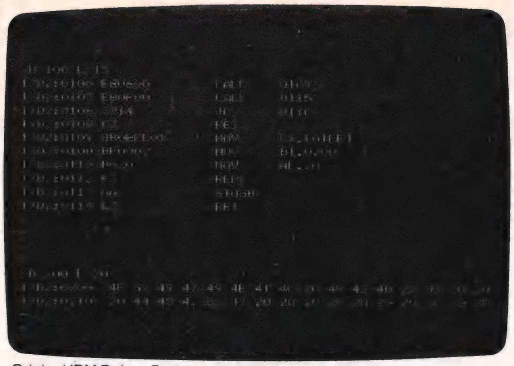
The one surprise in store for expression lovers is a pleasant one. Recall that the first operand of an AND or OR operator determines whether or not it is necessary to evaluate the second operand (i.e., if the first operand of an AND is false, the result is false, and the value of the second operand is unimportant). In

```
TYPE CommandProcedure = PROCEDURE(ch: CHAR);
VAR Command: ARRAY CHAR OF CommandProcedure;
        TimeToQuit: BOOLEAN;
PROCEDURE CommandInterpreter( );
VAR ch: CHAR;
BEGIN
    TimeToQuit := FALSE;
    REPEAT
        Terminal. Read(ch);                 (* get a character from key *)
        Command[ch] (ch);                   (* call command routine     *)
    UNTIL TimeToQuit;
END CommandInterpreter;
. .
PROCEDURE InitializeCommandTable;
BEGIN
    FOR ch := 0C TO 177C DO
        Command[ch] := IllegalCommand
    END;
    Command["q"] := QuitCommand;
    . . .
END InitializeCommandTable;
PROCEDURE IllegalCommand(ch: CHAR);
BEGIN
    Terminal. Write(ch); Terminal. Write("?");
    Terminal. WriteLn;
END IllegalCommand;
PROCEDURE QuitCommand(ch: CHAR);
BEGIN
    TimeToQuit := TRUE
END QuitCommand;
```

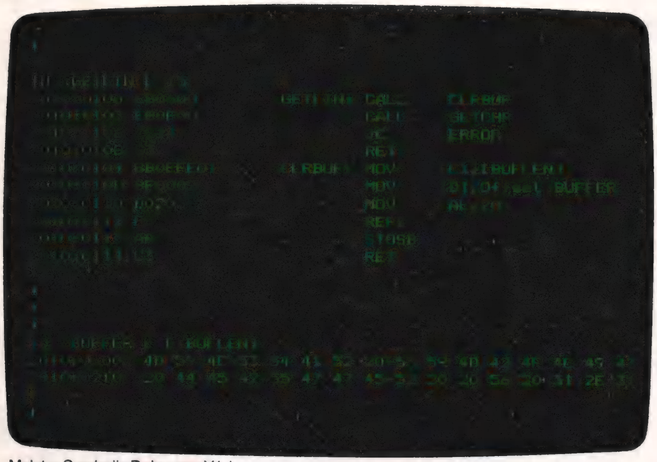
Figure 4.

Modula-2 code illustrating procedures

The difference



Original IBM Debug Program



Mylstar Symbolic Debugger V1.1

is Mylstar's Symbolic Debugging Program*

The plain and simple difference is that Mylstar's Symbolic Debugging Program speaks to your IBM PC in a language you both can understand, plain and simple.

Employing the same command structure, it allows you to use symbol names, mathematical expressions, batch files, on-line help, multi-command macros and other time-saving entries.

It's the enhancement to the *IBM Debug Program* you've been looking for—because it fills in the gaps—shortening the frustrating debugging process by as much as 50%—leaving you more time to do the work you need to do and the work you want to do, plain and simple.

Mylstar's Symbolic Debugging Program has been programmer-tested for over a year at Mylstar Electronics, Inc., (formerly D. Gottlieb & Co.), designers of the video arcade game, Q*BERT™.

TO ORDER...
Call (312) 562-7400 or mail coupon today.

**Designed for IBM PC-DOS 1.1 with 128K RAM minimum*

MYLSTAR ELECTRONICS INC.

165 West Lake Street
Northlake, Illinois 60164

A Columbia Pictures Industries Company

Mylstar Electronics, Inc.,
165 W. Lake St., Northlake, IL 60164

Please send me Mylstar's Symbolic Debugging Program for use with the IBM PC computer. Enclosed is $125, plain and simple.

☐ Check ☐ Money Order 118

NAME

FIRM

ADDRESS

CITY STATE ZIP

Illinois residents add 7% sales tax
Allow 2-4 weeks for delivery

Circle **no. 43** on reader service card.

this case, not evaluating the second operand is called "short circuiting" the expression, and this is exactly what Pascal did *not* promise to do – but Modula-2 does.

If you don't see how important this is, consider the problem of searching a linked list for a record with a given key, where the key may not be present in the list. In Pascal, because it is not defined whether both operands of an AND are evaluated, we must write the loop very carefully to avoid accessing through a NIL pointer (Figure 6a, below). The corresponding loop in Modula-2 is more straightforward (Figure 6b, below).

Statements

One of the first things you are likely to notice when reading a Modula-2 program is the relative scarcity of the keyword BEGIN. Most statements in Modula-2 include an explicit terminator (usually END) and act on a sequence of statements, instead of on a single statement as in Pascal (although the single Pascal statement is often a sequence of statements inside of a BEGIN/END pair).

Modula-2 supports the same basic iteration and flow control structures as Pascal, with a few additions and one notable exception: there is no GOTO in Modula-2.

As shown in Figure 7 (below), the WHILE/DO is now terminated with END, and the REPEAT/UNTIL is absolutely unchanged. (In some sense, it was the most Modula-like statement in Pascal.)

To this set of iteration constructs, Modula-2 adds LOOP (Figure 8, below). Statements within a LOOP are executed until a RETURN or EXIT statement is encountered. The EXIT statement is only legal within the scope of a LOOP and terminates the nearest enclosing LOOP.

The Pascal FOR statement survives in Modula-2 with a slight increase in power. The DOWNTO form is eliminated, and a BY clause is allowed, to specify the step on each iteration. The default step is 1, of course, and the FOR statement is terminated with a matching END (Figure 9, below).

As you might expect, the IF statement requires an explicit terminating END, which is a little tedious but eliminates the problem of matching an ELSE with the correct IF clause. (The Modula-2 ELSIF construction provides some relief from the pain of IF-END nesting.) Figure 10 (page 27) demonstrates how this can remove possible problems.

The CASE statement has undergone the most change. Most notably, an ELSE clause has been added to catch missing values. This, of course, is present in one form or another in many Pascal implementations.

Case labels must still be constants, but ranges are allowed as well as single values. Also Modula-2 allows constant expressions where Pascal allows only a literal or constant identifier. The sequence of statements following a case label is separated from the next case label by a vertical bar (Figure 11, page 27).

Modules

The compilation unit in Modula-2 is a module. In general, as explained earlier, modules are split into two parts (a definition part and an implementation part), which are compiled independently. The definition part defines the interface that the module presents to the outside world: the "what" of the module. The implementation part (the "how") is intentionally hidden. To see how this works, consider a simple stack mechanism (Figure 12, page 27).

In addition to push and pop (the primary stack operators), the stack module exports two variables, Overflow and Underflow, through which errors are signaled. A module may export procedures, constants, types, and variables.

Note that the two variables declared

```
PROCEDURE Upcase(VAR str: ARRAY OF CHAR);
VAR i: CARDINAL;
BEGIN
    FOR i := 0 TO HIGH(str) DO
        str[i] := CAP(str[i])
    END
END Upcase;
```

Figure 5.
Array parameters in Modula-2

```
found := FALSE
WHILE (p <> NIL) AND NOT found DO BEGIN
    IF p^. key = k THEN
        found := TRUE
    ELSE
        p := p^. next
END;
```

Figure 6a.
Linear searching in Pascal

```
WHILE (p <> NIL) AND (p^. key <> k) DO
    p := p^. next
END;
```

Figure 6b.
The same fragment in Modula-2

```
WHILE (column < 80) AND (Card[column] = "0") DO
        Card[column] := ' ';
        column := column + 1;
END
REPEAT
        p := p^. next;
        length := length + 1;
UNTIL p = NIL
```

Figure 7.
The Modula-2 WHILE and REPEAT constructs

```
LOOP
        ReadLine( );
        IF eof EXIT;
        WriteLine( );
END
```

Figure 8.
The Modula-2 LOOP construct

```
FOR c := (Width - 1) TO Cursor BY -1 DO
        Line[c + 1] := Line[c];
END
```

Figure 9.
The Modula-2 FOR construct

in the definition part are not redeclared in the implementation part. This is because all constants, variables, and types declared in a definition part are known implicitly in the corresponding implementation part.

The third item of interest is the code at the bottom of the implementation module, where the top level of a Pascal program would be found. This code is called the module body, and it can be used to initialize the module. It is executed after the bodies of any imported modules are executed. In case of circular reference, no order is defined.

The main (top level) module of a Modula-2 program is simply a MODULE. It may (and most likely will) import definitions, but it does not export any.

How does Modula-2 ensure that the correct implementation modules are used when the program is linked? Each time a definition part is compiled, it is assigned a timestamp. When any other module is compiled that refers to that particular interface, the timestamp is included in the compiled code. Both the implementation of the interface and the clients of the interface receive these timestamps.

When modules are linked together, all of the timestamps for each interface are compared; if they are not identical, then a warning is issued. This happens if two modules are compiled with different versions of some definition part, which means that there could be an inconsistency between the definitions of objects and their use.

This section is not a comprehensive discussion of modules. We refer the interested reader to the Modula Report (in the back of *Programming in Modula-2*), particularly for further discussion of modules and low-level programming features. David L. Parnas has written several fundamental papers on the subjects of information hiding and the use of modules in the design of software systems. Doug Cooper's book, *Standard Pascal – User Reference Manual*, is a good reference for standard Pascal.

Finis

This is by no means the complete story on Modula-2. As we mentioned, we did not discuss certain provisions for concurrent programming and for low-level systems programming at all. We hope that the Pascal programmer who was curious about Modula-2 has had some questions answered and perhaps some new ones raised.

We are particularly indebted to the *DDJ* reviewers whose sharp questions and knowledgeable comments made this article much better than it would have been.

Hugh McLarty is manager of the Modula-2 Software Engineering Group at Logitech, Inc. in Palo Alto. David Smith is presently with Tolerant Systems, Inc.

DDJ


```
IF Total > 2000. 0 THEN
    IF Total > 10000. 0 THEN
        Total := Total + BigBonus
ELSE
    Total := Total + SmallBonus;
```

Figure 10a.

Incorrectly indented Pascal

```
IF Total > 2000. 0 THEN
    IF Total > 10000. 0 THEN
        Total := Total + BigBonus
    ELSE
        Total := Total + SmallBonus
     END
 END
```

Figure 10b.

The same rewritten in Modula-2

```
CASE ch OF
   ' ':                   Token := Blank
 | '0' . . '9':           Token := Number
 | 'A' . . 'F', 'a' . . 'z':  Token := Letter
                  Token := Letter
ELSE  Token  :=  Unknown
END
```

Figure 11.

A Modula-2 CASE statement

```
DEFINITION MODULE IntStack;
    EXPORT Push, Pop, Overflow, Underflow;
    VAR Overflow, Underflow : BOOLEAN;
    PROCEDURE Push (Item : INTEGER);
    PROCEDURE Pop ( ) : INTEGER;
END IntStack.
IMPLEMENTATION MODULE IntStack;
    CONST StackSize = 100;
    VAR Stk : ARRAY [1 . . STACKSIZE] OF INTEGER;
        Top : [0 . . STACKSIZE] ;
    PROCEDURE Push (Item : INTEGER);
    BEGIN
        IF Top = StackSize THEN
            Overflow := TRUE;
        ELSE
            Top :=  Top + 1;
            Stk[Top] = Item;
        END
    END Push;
    PROCEDURE Pop( ) : INTEGER;
    BEGIN
        IF Top = 0 THEN
            Underflow := TRUE;
            RETURN 0;
        ELSE
            Top = Top - 1;
            RETURN Stk[Top + 1] ;
    END Pop;
BEGIN (* Initialize the stack *)
    Top := 0;
    Overflow := FALSE;
    Underflow := FALSE;
END IntStack.
```

Figure 12.

Stack mechanism in Modula-2

Converting Fig-Forth to Forth-83

Forth has a colorful history compared to most modern computer languages. It evolved into its current form within a few years as the result of the work of one man (Charles Moore), was jealously guarded as a proprietary product for quite a few more years, and came into widespread use only after a group of volunteer systems programmers created a set of public domain implementations in 1978. The same programmers founded the Forth Interest Group (FIG) shortly thereafter.

Flushed by the success of the FIG implementations, our heroes formed the Forth Standards Team. The team was charged with creating a clear specification for the Forth language that could be used as a reference by both users and systems vendors. Ignoring the old engineering adage, "If it works, don't fix it," they produced the document known as the Forth-79 Standard, which was widely criticized and then almost as widely ignored.

Back to the drawing boards for the Forth Standards Team. Reconvening in late 1982, they began the traditionally grueling process of creating a new Standard, which involves much lobbying, arguing, and infighting. In August 1983, they took a final vote and approved the Forth-83 Standard document.

Even before Forth-83 hit the streets, it drew hostile comments from many vendors because it ignored or bypassed a number of important issues (such as reading device status and 32-bit implementations), was exceedingly vague about vocabularies (considered a vital feature of the language), and actually redefined the action of certain keywords and control structures that had been in common use for years – in essence creating a new language incompatible with all existing Forth programs.

The reader may well ask how such a Standards document could be created and approved. The politics of Forth would make a fascinating subject for a sociologist's dissertation, but I'm not sure I understand them well enough to explain them. For the purposes of this article, let us just note that both the board of directors of FIG and the members of the Forth Standards Team are self-elected. Nearly all have been in office since the beginning of FIG, and their responsiveness and accountability to the membership of FIG (let alone the general community of Forth vendors and users) is minimal.

With all of its peculiarities, the Forth-83 Standard seems likely to come into widespread use fairly quickly. The alternative is to stick with Forth-79, which has major deficiencies, or with fig-FORTH (Forth-78), which has become very dated. Two of the largest Forth vendors (Laboratory Microsystems and Micromotion) have already committed to converting all of their systems to Forth-83 and have independently created two rather similar implementations.

A third, public domain implementation, known as F83 and written by Michael Perry and Henry Laxen, has also become available through certain user groups. It is a powerful system that includes a full screen editor, assembler, and other programming tools. Unfortunately, F83 consists of a very large and confusing body of source code that incorporates many cute Forth programming "tricks" and shortcuts. I doubt that Forth beginners will find the F83 model comprehensible.

The remainder of this article consists of a detailed guide to conversion of existing fig-FORTH software to Forth-83. The instructions assume that the target system is a Laboratory Microsystems (LMI) or Micromotion implementation, though the instructions should be applicable for the most part to any Forth-83 system. The material assumes an average level of Forth programming competence and is not directed to readers unfamiliar with the Forth language.

by Ray Duncan

Ray Duncan, Laboratory Microsystems, P. O. Box 10430, Marina del Rey, CA 90295.

Conversion Overview

Due to the Forth Standard Team's lack of concern for upward compatibility with the commonly available fig-FORTH, poly-FORTH,™ or Forth-79 systems, it is quite likely that your programs will need some careful inspection and editing before they will execute properly on top of a Forth-83 Standard nucleus; we have provided a checklist of things to look for. Because this is not guaranteed to be inclusive, refer to the 83-Standard document and the glossary section of your Forth System User Manual for more detailed information. In case of a conflict between the two, assume the 83-Standard document description to be correct.

1. Due to the runtime requirements of the redefined **LOOP**, **+LOOP**, and **LEAVE** words, the return stack is maintained in a different format than in previous implementations. In LMI Forth systems, the **LOOP** or **+LOOP** construct requires three control words on the return stack. Programs that rely on specialized knowledge of the return stack will require extensive changes and cannot, in any case, be considered "standard" or "portable."

2. All "state smart" words such as ' (tick) and ." (dot-quote), which previously had different actions depending on whether they were invoked inside or outside a colon definition, have been either eliminated or redefined.

3. The fig-FORTH words **CFA**, **PFA**, **LFA**, and **NFA**, which were used to find the addresses of different fields within a dictionary header, have been eliminated. A new set of words, adopted from the Kim Harris experimental proposal, has been included in the new LMI and Micromotion systems:

BODY>	**LINK>**
>BODY	**>LINK**
NAME>	**N>LINK**
>NAME	**L>NAME**

Detailed explanations will come in a later section. At present, the only word of this set that is part of the 83-Standard is **>BODY**.

4. For various reasons the definition of all divide functions (except for the unsigned divide) has been changed slightly. The general effect is that quotients are floored instead of rounded toward zero. This should cause no problems for most preexisting application software. The new divide functions are marginally slower than the old (a few machine cycles under most circumstances). The side effects of redefinition for floored divide can be counter-intuitive under some circumstances. For example, in fig-FORTH the operation

-40 360 MOD

would return the obvious answer (–40) on the stack, while Forth-83 returns 320!

5. The true flag returned by all logical operations has been changed from the value 1 to the value -1 (all bits set). If your code uses 0 or 1 returned by a comparison in an arithmetic operation, you will need to interpolate the operator **ABS** after the logical operator. This is a particularly difficult problem to look for in your source code. However, I feel that this mutation in the 83-Standard was beneficial as it allows the returned true/false value to be used as a mask for **AND**.

6. **PICK** and **ROLL** are now zero-based, instead of one-based. The reasoning behind this change uses the rather weak argument that programmers frequently use zero as the beginning index for loops.

7. The word **LEAVE** has become an "Immediate" compiler word with state-checking to satisfy the 83-Standard's requirements. It, in turn, compiles the runtime word **(LEAVE)**. If your previous code includes compiler extensions referencing **LEAVE**, it may need modification.

In addition, the runtime action of **LEAVE** is immediate; that is, it causes a direct transfer of control past the end of the currently active innermost loop. Consequently, the new **LEAVE** is unstructured, so that a loop increment may accidentally be passed outside the loop construct under certain circumstances. This problem is easier to demonstrate than explain; observe the following (admittedly contrived) example:

```
: TEST 1000 0 DO 2 ?TERMINAL
IF LEAVE THEN +LOOP ;
```

If **?TERMINAL** returns a true flag, after exit from the **DO . . . +LOOP** construct, 2 will remain on the stack in the Forth-83 system whereas it would have been discarded in fig-FORTH.

8. Certain very commonly used Forth words were not included in the Standard and their eventual fate should be regarded as uncertain: they might be standardized with a different definition or abolished altogether. The list includes:

S->D	OUT	ERROR
?TERMINAL	SCR	?ERROR
CONTEXT	DPL	MESSAGE
CURRENT	HLD	.LINE
FENCE	R#	THRU
DP	C,	-->

Restrictions on an 83-Standard Forth Application Program

Forth systems, whether complying with any Standard or not, typically contain many environmentally dependent words in addition to the Standard's required word set. In order for your application programs to be portable to any Standard Forth system, you should observe the following rules (abridged from the Standard document).

1. A Standard application may reference only the definitions of the 83-Standard Required Word Set and Standard Extensions and any definitions that are subsequently defined in terms of those words.

2. A Standard program may operate only on data that was stored by the application. The initial contents of variables and arrays created at compilation time are explicitly undefined.

3. A Standard program may address:

Parameter fields of words created with **CREATE**, **VARIABLE**, and user-defined words that execute **CREATE**
Dictionary space **ALLOT**ted
Data in a valid mass storage buffer
Data area of user variables
Text Input Buffer (**TIB**) and **PAD** up to amount specified as the minimum for each area

4. A Standard program may not address:

Directly into the data or return stacks
Into a definition's name, link, or code fields
Into a definition's parameter field if its contents were not stored by the application

Although the capability of doing these types of operations is probably present in the system you are using (**SP@**, **RP@**, **>NAME**, etc.), the operations should be avoided if you intend to port your program to 83-Standard systems provided by different vendors.

Program Conversion to Forth-83

This checklist is not foolproof, but it can save you a lot of time just getting your programs to the point where they will compile (proper execution, of course, is another problem). Most of the changes indicated are related to adoption of the Forth-83 Standard; other renamings are not properly part of the Standard but have been adopted in the Laboratory Microsystems, Micromotion, or public domain Laxen/Perry models.

1. Search for all instances of **R** ; replace with **R@**.

2. Search for all instances of **-DUP** ; replace with **?DUP**.

3. Search for all instances of **WORD**. When it occurs as **WORD HERE**, delete the **HERE**. When **WORD** is not followed by **HERE**, replace it with **WORD DROP**.

4. Search for all instances of **PICK** and **ROLL** ; replace them by **1- PICK** and **1- ROLL**, respectively.

5. Examine all **DO . . . LOOP**s. For any loop that might be entered with the limit equal to the index, replace **DO** with **?DO**.

6. Search for all instances of **LEAVE**. Note that the action of **LEAVE** will be immediate. If it occurs within an **IF . . . ELSE . . . THEN** clause, **LEAVE** should be the last word before the **ELSE** or **THEN**. If **LEAVE** is used within a **DO . . . +LOOP** construct, make sure that the incrementing value will not remain on the stack if **LEAVE** is executed.

7. Search for all instances of ' (tick) within a colon definition. If it is not preceded by **[COMPILE]**, replace it with **[']**.

8. Search for all instances of **."** (dot-quote) outside of a colon definition; replace the **."** with **.(** and the closing delimiter **"** with **)** to prevent compilation failures.

9. Search for all instances of **NFA**, **PFA**, **LFA**, and **CFA**. It is best to examine these and recode them individually, but you can make some "brute force" substitutions as follows:

This:	*Becomes:*
'	**' >BODY** (outside colon definition)
' CFA	**'** (outside colon definition)
' CFA	**[']** (inside colon definition)
NFA	**BODY> >NAME**
' NFA	**' >NAME**
LFA	**BODY> >LINK**
' LFA	**' >LINK**
PFA	**NAME> >BODY**
PFA CFA	**NAME>**

Bear in mind that the old definitions had the following actions:

CFA	(pfa --- cfa)
NFA	(pfa --- nfa)
LFA	(pfa --- lfa)
PFA	(nfa --- pfa)

These were predicated on the fact that ' returned the parameter field address. Since ' and **[']** now return the code field address, the new words suggested by Kim Harris revolve around that value:

>BODY	(cfa --- pfa)
>LINK	(cfa --- lfa)
>NAME	(cfa --- nfa)
BODY>	(pfa --- cfa)
LINK>	(lfa --- cfa)
NAME>	(nfa --- cfa)

These are appealing and symmetric, but before you get too carried away with them remember that a Standard program can't access any part of a dictionary definition except for the parameter field ("body"). Two additional words are provided for convenience in traversing the linked dictionary list:

N>LINK	(nfa --- lfa)
L>NAME	(lfa --- nfa)

10. Search for all instances of **ENDIF** and replace with **THEN**.

11. Search for all instances of **MINUS** or **DMINUS** and replace with **NEGATE** or **DNEGATE**, respectively.

12. Search for all instances of **SIGN** and

fix up stack logic. Usually **SIGN** can be replaced by **ROT SIGN**.

13. Any use of **-FIND** must be recoded individually. You can usually replace it with the sequence **BL WORD FIND**.

14. Search for all instances of ? and replace with @ . .

15. Search for all instances of **BLANKS**; replace with **BLANK**.

16. Use of old **CREATE** words in your programs must be modified. In most instances on LMI implementations, you can replace it with the new word **BUILD**. The Forth-83 word **CREATE** has a much different effect.

17. Search for all instances of the construct **<BUILDS . . . DOES>** and replace with **CREATE . . . DOES>**.

18. Search for all instances of **END** and replace with **UNTIL**.

19. Search for all instances of **(NUMBER)** and replace with **CONVERT**.

20. Search for all instances of **IN** and replace with **>IN**.

21. Search for all instances of **FLUSH** and replace with **SAVE-BUFFERS**.

22. Search for all instances of **U*** and replace with **UM***; similarly, replace all occurrences of **U/** with **UM/MOD**.

23. Replace all instances of **S->D** with **S>D**.

24. Replace the word **SP!** with the sequence **S0 @ SP!**, and change the word **RP!** to **R0 @ RP!**.

25. Search for all instances of **TIB @** and replace with **TIB**. Recode any sequences of **TIB!** in an implementation-dependent manner.

26. Difficult to find by inspection but very dangerous is the use of the Boolean flag returned by a comparison in a calculation, such as the sequence **0= ADD**. These must be recoded individually.

27. Also difficult to find is the use of specialized knowledge of the return stack (such as fetching the index of the third outer loop). These must be individually examined and recoded.

28. Search for all **VARIABLE** declarations and delete leading initializing value (these do no harm but will be left as residual data on the stack at the end of compilation). Although most implementations do set the initial value of a variable to zero or -1, a Standard application program is, of course, not allowed to take advantage of this information. It is most correct to initialize variables at runtime (so that the code is reusable); however, your old fig-FORTH compile-time initialization of variables can easily be mimicked. For example, the fig-FORTH statement

4 VARIABLE XVAR

would be changed to

VARIABLE XVAR 4 XVAR !

29. Search for all instances of **+-** and **D+-**; replace them with **?NEGATE** and **?DNEGATE**, respectively.

30. Examine occurrences of **MOD**. If either argument can take on a negative value, the results may surprise you.

31. Find all **M/MOD** and replace with **MU/MOD**. Search for all instances of **M/**; replace with **M/MOD**.

References

Forth-83 Standard. A publication of the Forth Standards Team. Available for $15.00 from Laboratory Microsystems Inc., P. O. Box 10430, Marina del Rey, California 90295.

FORTH TOOLS, Volume 1. By Anita Anderson and Martin Tracy. A new tutorial on Forth; all example programs are written in Forth-83. Available for $20.00 from Micromotion, 12077 Wilshire Blvd. #506, Los Angeles, California 90025.

"Signed Integer Division." By Robert L. Smith. *Dr. Dobb's Journal*, September 1983, page 86.

"When a Page Is Not a Page: Forth-83 and Vocabularies." By George William Shaw II. *Dr. Dobb's Journal*, September 1983, p. 90.

DDJ

(A Summary of Vocabulary Changes begins on page 32)

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 193.**

Summary of Vocabulary Changes in Forth-83

#TIB	A new 83-Standard user variable containing the length in bytes of the valid input stream within the terminal input buffer. This is *not* the length of the buffer itself. **#TIB** is set by **QUERY** after calling **EXPECT**.
	"Tick" is no longer "immediate" and returns the CFA of the target name instead of the PFA as in previous versions. Outside a colon definition, your previous code **' CFA** should be replaced by **'** while inside a colon definition it should be replaced by **[']**.
(FIND)	Deleted. To perform the function of the old **(FIND)**, use the new word **FIND**, which is 83-Standard.
+LOOP	Termination has been redefined to occur when the INDEX crosses the boundary between LIMIT-1 and LIMIT. This is implemented in LMI Forth by adjusting LIMIT to 8000H and checking for a machine overflow flag after incrementing the INDEX. This **+LOOP** is faster than in the previous version but behaves somewhat differently. For example: **1 1 DO . . . 1 +LOOP** will execute 65,536 times (in fig-FORTH or Forth-79, it would have executed only once) while **1 1 DO . . . -1 +LOOP** will execute once. See also **LOOP**.
+-	Renamed to **?NEGATE** as suggested by Laxen.
+ORIGIN	Deleted.
-DUP	Renamed to **?DUP**.
-FIND	Essentially replaced by the 83-Standard word sequence **BL WORD FIND**.
."	"Dot-quote" now may be used inside of colon definitions only. See **.(** also.
.(	New word. Immediate. The characters up to but not including the delimiting **)** are displayed on the standard output device (usually the operator's console). May be used inside or outside of a colon definition.
.NAME	Performs the function of the old **ID.**.
<BUILDS	Replaced by **CREATE**, which has some slightly different effects.
<CMOVE	Renamed to **CMOVE>** in 83-Standard Forth.
>BODY	Converts code field address to parameter field address.
>LINK	Converts code field address to link field address. Not 83-Standard.
>NAME	Converts code field address to name field address. Not 83-Standard.
?	Deleted.
?DNEGATE	New name for **D+-** as suggested by Laxen. Not 83-Standard.
?DO	Works like **DO** except executes zero times if the input INDEX and LIMIT are equal. A word suggested by Laxen/Harris/Perry. Not 83-Standard, but present in LMI and Micromotion systems.
?DUP	Same as old **-DUP**.
?NEGATE	New name for **+-** as suggested by Laxen. Not 83-Standard.
ABORT"	A new word that requires a flag on top of stack; it does nothing if the flag is false and prints a string and executes **ABORT** if the flag is true. Used like **?ERROR** but with no requirement for a disk access.
AGAIN	Still present in LMI Forth systems with function unchanged, but not present in the 83-Standard or even in the Controlled Reference Word Set. Probably fated for extinction, so its use should be avoided in new code. Can be replaced by **0 UNTIL**.
BLANK	Renamed from **BLANKS**. See below.
BLANKS	Renamed to **BLANK** (not an 83-Standard word, but included in the Controlled Reference Word Set).
BODY>	Converts parameter field address to code field address. Works like the old word **CFA**. Not 83-Standard.
CFA	Removed. See **BODY>**, **NAME>**, and **LINK>**.
CMOVE>	Previously known as **<CMOVE**.
CONVERT	Essentially works like the old fig-FORTH word **(NUMBER)**. Caution: converts positive double numbers only. Sign handling must be done outside.
CREATE	Redefined from fig-FORTH. Now used with **DOES>** in the same manner as the old **<BUILDS**.

(Continued on page 34)

Z80 Software

DEBUG QED LINKER MODEM MACRO TPM BASIC I ZDDT ZTEL QSAL ZEDIT TOP I 6502X TPM II

SOFTWARE DESCRIPTIONS

TPM (TPM I) - $80 A Z80 only operating system which is capable of running CP/M programs. Includes many features not found in CP/M such as independent disk directory partitioning for up to 255 user partitions, space, time and version commands, date and time, create FCB, chain program, direct disk I/O, abbreviated commands and more! Available for North Star (either single or double density), TRS—80 Model I (offset 4200H) or II, Versafloppy I, or Tarbell I.

TPM-II - $125 An expanded version of TPM which is fully CP/M 2.2 compatible but still retains the extra features our customers have come to depend on. This version is super FAST. Extended density capability allows over 600K per side on an 8" disk. Available preconfigured for Versafloppy II (8" or 5"), Epson QX-10, Osborne II or TRS-80 Model II.

CONFIGURATOR I
This package provides all the necessary programs for customizing TPM for a floppy controller which we do not support. We suggest ordering this on single density (8SD).
Includes: TPM-II ($125), Sample PIOS (BIOS) SOURCE ($FREE), MACRO II ($100), LINKER ($80), DEBUG I ($80), QED ($150), ZEDIT ($50), TOP I ($80), BASIC I ($50) and BASIC II ($100)
$815 Value **NOW $250**

CONFIGURATOR II
Includes: TPM-II ($125), Sample PIOS (BIOS) SOURCE ($FREE), MACRO II ($100), MACRO III ($150), LINKER ($80), DEBUG I ($80), DEBUG II ($100), QSAL ($200), QED ($150), ZTEL ($80), TOP II ($100), BUSINESS BASIC ($200) and MODEM SOURCE ($40) and DISASSEMBLER ($80)
$1485 Value **NOW $400**

MODEL I PROGRAMMER
This package is only for the TRS-80 Model I. Note: These are the ONLY CDL programs available for the Model I. It includes: TPM I ($80), BUSINESS BASIC ($200), MACRO I ($80), DEBUG I ($80), ZDDT ($40), ZTEL ($80), TOP I ($80) and MODEM ($40)
$680 Value **NOW $175**

MODEL II PROGRAMMER
This package is only for the TRS-80 Model II. It includes: TPM-II ($125), BUSINESS BASIC ($200), MACRO II ($100), MACRO III ($150), LINKER ($80), DEBUG I ($80), DEBUG II ($100), QED ($150), ZTEL ($80), TOP II ($100), ZDDT ($40), ZAPPLE SOURCE ($80), MODEM ($40), MODEM SOURCE ($40) and DISASSEMBLER ($80)
$1445 Value **NOW $375**

BASIC I - $50, a 12K+ basic interpreter with 7 digit precision.

BASIC II - $100, A 12 digit precision version of Basic I.

BUSINESS BASIC - $200, A full disk extended basic with random or sequential disk file handling and 12 digit precision (even for TRIG functions). Also includes PRIVACY command to protect source code, fixed and variable record lengths, simultaneous access to multiple disk files, global editing, and more!

ACCOUNTING PACKAGE - $300, Written in Business Basic. Includes General Ledger, Accounts Receivable/Payable, and Payroll. Set up for Hazeltine 1500 terminal. Minor modifications needed for other terminals. Provided in unprotected source form.

MACRO I - $80, A Z80/8080 assembler which uses CDL/TDL mnemonics. Handles MACROs and generates relocateable code. Includes 14 conditionals, 16 listing controls, 54 pseudo-ops, 11 arithmetic/logical ops, local and global symbols, linkable module generation, and more!

MACRO II - $100, An improved version of Macro I with expanded linking capabilities and more listing options. Also internal code has been greatly improved for faster more reliable operation.

MACRO III - $150, An enchanced version of Macro II. Internal buffers have been increased to achieve a significant improvement in speed of assembly. Additional features include line numbers, cross reference, compressed PRN files, form feeds, page parity, additional pseudo-ops, internal setting of time and date, and expanded assembly-time data entry.

DEVELOPER I
Includes: MACRO I ($80), DEBUG I ($80), ZEDIT ($50), TOP I ($80), BASIC I ($50) and BASIC II ($100)
$440 Value **NOW $150**

DEVELOPER II
Includes: MACRO II ($100), MACRO III ($150), LINKER ($80), DEBUG I ($80), DEBUG II ($100), BUSINESS BASIC ($200), QED ($150), TOP II ($100), ZDDT ($40), ZAPPLE SOURCE ($80), MODEM SOURCE ($40), ZTEL ($80), and DISASSEMBLER ($80).
$1280 Value **NOW $350**

DEVELOPER III
Includes: QSAL ($200), QED ($150), BUSINESS BASIC ($200), ZTEL ($80) and TOP II ($100)
$730 Value **NOW $300**

COMBO
Includes: DEVELOPER II ($1280), ACCOUNTING PACKAGE ($300), QSAL ($200) and 6502X ($150)
$1930 Value **NOW $500**

LINKER - $80, A linking loader for handling the linkable modules created by the above assemblers.

DEBUG I - $80, A tool for debugging Z80 or 8080 code. Disassembles to CDL/TDL mnemonics compatible with above assemblers. Traces code even through ROM. Commands include Calculate, Display, Examine, Fill, Goto, List, Mode, Open File, Put, Set Wait, Trace, and Search.

DEBUG II - $100, A superset of Debug I. Adds Instruction Interpreter, Radix change, Set Trap/Conditional display, Trace options, and Zap FCB.

6502X - $150, A 6502 cross assembler. Runs on the Z80 but assembles 6502 instructions into 6502 object code! Similar features as our Macro assemblers.

QSAL - $200, A SUPER FAST Z80 assembler. Up to 10 times faster than conventional assemblers. Directly generates code into memory in one pass but also to offset for execution in its own memory space. Pascal like structures: repeat...until, if...then...else, while...do, begin...end, case...of. Multiple statements per line, special register handling expressions, long symbol names, auto and modular assembly, and more! This one uses ZILOG Mnemonics.

QED - $150, A screen editor which is both FAST and easy to learn. Commands include block delete, copy, and move to a named file or within text, repeat previous command, change, locate, find at start of line, and numerous cursor and window movement functions. Works with any CRT having clear screen, addressable cursor, clear to end of line, clear to end of screen, and 80X24.

ZTEL - $80, An extensive text editing language and editor modelled after DEC's TECO.

ZEDIT - $50, A mini text editor. Character/line oriented. Works well with hardcopy terminals and is easy to use. Includes macro command capability.

TOP I - $80, A Text Output Processor for formatting manuals, documents, etc. Interprets commands which are entered into the text by an editor. Commands include justify, page number, heading, subheading, centering, and more.

TOP II - $100, A superset of TOP I. Adds: embedded control characters in the file, page at a time printing, selected portion printing, include/merge files, form feed/CRLF option for paging, instant start up, and final page ejection.

ZDDT - $40, This is the disk version of our famous Zapple monitor. It will also load hex and relocatable files.

ZAPPLE SOURCE - $80, This is the source to the SMB ROM version of our famous Zapple monitor. It can be used to create your own custom version or as an example of the features of our assemblers. Must be assembled using one of our assemblers.

MODEM - A communication program for file transfer between systems or using a system as a terminal. Based on the user group version but modified to work with our SMB board or TRS-80 Models I or II. You must specify which version you want.

MODEM SOURCE - $40, For making your own custom version. Requires one of our Macro Assemblers.

DISASSEMBLER - $80, Does bulk disassembly of object files creating source files which can be assembled by one of our assemblers.

HARDWARE

S-100 — SMB II Bare Board $50, "System Monitor Board" for S-100 systems, 2 serial ports, 2 parallel ports, cassette interface, 4K memory (ROM, 2708 EPROM, 2114 RAM), and power on jump. When used with Zapple ROM below, it makes putting a S-100 system together a snap.
Zapple ROM $35, Properly initializes SMB I/II hardware, provides a powerful debug monitor.

IBM PC — Big Blue Z80 board $595, Add Z80 capability to your IBM Personal Computer. Runs CP/M programs but does not require CP/M or TPM. Complete with Z80 CPU, 64K add on memory, serial port, parallel port, time and date clock with battery backup, hard disk interface, and software to attach to PC DOS and transfer programs. Mfr'd by QCS.
50% Discount on all CDL software ordered at the same time as a Big Blue (and for the Big Blue).

APPLE II — Chairman Z80 $345, Add Z80 capability to your Apple II/II Plus computer. Runs CP/M programs with our more powerful TPM. Includes 64K memory add on (unlike the competition this is also useable by the 6502/DOS as well as the Z80), TPM, QSAL assembler, QED Screen Editor, and Business Basic. Mfr'd by AMT Research.
Apple Special $175, Buy the Apple Z80 Developer at the same time as the "Chairman" and pay only $175 instead of $325.

APPLE Z80 DEVELOPER
Includes: 6502X ($150), MACRO II ($100), MACRO III ($150), QSAL ($200), QED ($150), LINKER ($80), DEBUG I ($80), DEBUG II ($100), ZDDT ($40) and BUSINESS BASIC ($200)
VALUE: $1250 **NOW $325**
$175 when purchased with AMT "Chairman" Board

DISK FORMATS

When ordering software specify which disk format you would like.

CODE	DESCRIPTION
8SD	8" IBM 3740 Single Density (128 bytes/26 sectors/77 tracks)
8DD	8" Double Density (256 bytes/26 sectors/77 tracks)
8XD	8" CDL Extended Density (1024 bytes/8 sector/77 traceks = 616K)
5SD	5.25" Single Density (TRS80 Model I, Versafloppy I, Tarbell I)
5EP	5.25" Epson Double Density
5PC	5.25" IBM PC Double Density
5XE	5.25" Xerox 820 Single Density
5OS	5.25" Osborne Single Density
5ZA	5.25" Z80 Apple (Softcard compatible)

TPM INFO When ordering TPM I or II, in addition to Disk Format, please specify one of the following codes:

	CODE	DESCRIPTION
TPM I:	NSSD/H	North Star Single Density for Horizon I/O
	NSSD/Z	North Star Single Density for Zapple I/O
	NSDD/H	North Star Double Density for Horizon I/O
	NSDD/Z	North Star Double Density for Zapple I/O
	TRS80-I	TRS-80 Model I (4200H Offset)
	TRS80II	TRS-80 Model II
	VI8	Versafloppy I 8"
	VI5	Versafloppy I 5.25"
TPM-II:	VII8	Versafloppy II 8" (XD)
	VII5	Versafloppy II 5.25"
	TRS80II	TRS-80 Model II (XD)

Prices and Specifications subject to change without notice.
TPM, Z80, CP/M, TRS80 are trademarks of CDL, Zilog, DRI and Tandy respectively.

ORDERING INFORMATION:

VISA/MasterCard/C.O.D.
Call or Write With Ordering Information....

MasterCard VISA

OEMS:

Many CDL products are available for licensing to OEM's. Write to Carl Galletti with your requirements.

Dealer Inquiries Invited.

For Phone Orders ONLY Call Toll Free...
1-(800) 458-3491 (Except Pa.)
Ask For Extension #15

For information and Tech Queries call
(609) 599-2146

Computer Design Labs

342 Columbus Avenue/Trenton, NJ 08629

(Continued from page 32)

D+-	Renamed to **?DNEGATE** as suggested by Laxen.
DIGIT	Returns a true flag of -1 instead of 1 as in previous version. This is not an 83-Standard word.
DMINUS	Renamed to **DNEGATE** in 83-Standard systems.
DNEGATE	83-Standard word, previously called **DMINUS**.
DOES>	Functionally similar to before, but implementation is considerably different and involves a mixture of machine code and high-level code in the defining word's parameter field.
EMPTY-BUFFERS	Marks all disk block buffers as unassigned. Does *not* write blocks marked for **UPDATE** to the disk. Not an 83-Standard word but still present in LMI Forth systems. See also **SAVE-BUFFERS** and **FLUSH**.
END	Removed. Use **UNTIL**.
ENDIF	Removed. Use **THEN**.
EXPECT	Works about the same but leaves the actual length of the input in the system variable **SPAN**.
FIND	New word that essentially provides the capabilities of the old **-FIND** and "state-smart tick."
FLUSH	Writes all **UPDATE**d blocks to disk then unassigns all block buffers. See also **EMPTY-BUFFERS**, **SAVE-BUFFERS**.
ID.	Renamed to **.NAME**. Not 83-Standard.
IN	Renamed to **>IN**.
LEAVE	Causes an immediate transfer of control to the code just beyond the next **LOOP** word. For example, in the code: **DO . . . IF XXX ELSE LEAVE YYY THEN LOOP** if the **ELSE** path is taken, the word **YYY** will never be executed (unlike fig-FORTH or Forth-79).
L>NAME	Converts link field address to name field address. Not 83-Standard.
LINK>	Converts link field address to code field address. Not 83-Standard.
LITERAL	Compilation only. The actual run-time word compiled may depend on the magnitude of the literal value.
LOAD	Loading from screen 0 is now defined to be illegal.
LOOP	Termination has been redefined to occur when the INDEX crosses the boundary between LIMIT-1 and LIMIT. This is implemented in LMI Forth by adjusting LIMIT to 8000H and checking for a machine overflow flag after incrementing the INDEX. This **LOOP** is faster than in the previous version but behaves somewhat differently. For example: **1 1 DO . . . LOOP** will execute 65,536 times (in fig-FORTH or Forth-79, it would have executed only once).
M*	No longer present in 83-Standard or Controlled Reference Word Set. Still present in LMI Forth systems, but use should probably be avoided.
M/	Renamed to **M/MOD**. This is different than the **M/MOD** that was present in fig-FORTH (which function is now carried out by **MU/MOD** in some implementations).
MINUS	Renamed to **NEGATE** in 83-Standard systems.
MOVE	This word is listed in the Uncontrolled Reference Word Set of the 83-Standard with a different action than was common in fig-FORTH. Avoid it.
N>LINK	Converts name field address to link field address. Not 83-Standard.
NAME>	Converts name field address to code field address. Not 83-Standard.
NEGATE	83-Standard word, previously named **MINUS**.
NFA	Removed. See **>NAME**.
NOT	Returns 1's complement.
PAD	Now provides a work area of at least 84 bytes.
PICK	The argument to **PICK** is now zero-based (was previously one-based). Examples: **0 PICK** is equivalent to **DUP** and **1 PICK** is equivalent to **OVER**.
R@	New name for fig-FORTH **R**.
R	Renamed to **R@** in 83-Standard.
ROLL	The argument to **ROLL** is now zero-based, instead of one-based. Examples: **0 ROLL** is a null operation, **1 ROLL** is equivalent to **SWAP**, **2 ROLL** is equivalent to **ROT**, etc.
RP!	Takes its argument from the top of the data stack rather than **R0**. A change suggested by Laxen/Perry for symmetry with **RP@**. The word **RP!**

	in your previous code should be replaced with the sequence **R0 @ RP!**. Not 83-Standard.
S->D	Renamed to **S>D** as suggested by Laxen. Not 83-Standard.
SAVE-BUFFERS	All buffers marked as **UPDATE**d are written to the disk but remain assigned. See also **EMPTY-BUFFERS** and **FLUSH**.
SIGN	Definition changed: takes its argument from the top of stack rather than the third item on the stack. The word **SIGN** in your old code can be replaced by **ROT SIGN**.
SP!	Takes its argument from the top of stack rather than **S0**. A change suggested by Laxen/Harris/Perry for symmetry with **SP@**. The word **SP!** in your previous code should be replaced with **S0 @ SP!**. Not 83-Standard.
SPAN	A new system variable that contains the length of the last input via **EXPECT**.
STATE	System variable. In LMI Forth systems, the value of **STATE** is -1 if compiling, 0 otherwise. An 83-Standard application shouldn't modify this variable.
TIB	Definition changed. Formerly returned the address of the location containing the address of the terminal input buffer. Now returns the actual address of the buffer. In your existing code, replace all sequences **TIB @** by the word **TIB** alone.
U*	Renamed to **UM***.
U/	Renamed to **UM/MOD**.
UM*	New name for **U***.
UM/MOD	New name for **U/**.
VARIABLE	Does not accept an initializing value.
VLIST	Renamed to **WORDS** (suggestion of Laxen).
VOCABULARY	The 83-Standard, in an attempt to remedy the 79-Standard's restrictiveness on vocabulary structures, left most details to the imagination of the Forth implementor. A radically different experimental proposal was offered by Bill Ragsdale and printed as an appendix to the Standards document, but this proposal was not approved as part of the Standard proper and has not been adopted by all Forth vendors at this point. Beware!
VOCS	(LMI implementations) Displays the current search order and the names of all declared vocabularies in the system. Not 83-Standard.
WIDTH	No longer present in LMI implementations. The full name (up to 31 characters) is always used.
WORD	Defined slightly changed from fig-FORTH. Returns the address of the first byte of the token it scanned off the input stream. A token may be up to 255 bytes in length.
WORDS	Displays the names of the definitions in the vocabulary that is first in the current search order. Similar to old **VLIST** command. Not 83-Standard.
[']	Used inside a colon definition to compile the CFA of the following word, like the old "state-smart tick" word. See also **'** and **FIND**.

(End Table)


KIRBY

Sixth Generation Computers

Dr. Dobb's periodically prints material designed to provoke thought on topics which look toward the future. In December 1982, we published an essay by Richard Grigonis entitled "Fifth-Generation Computers," which sketched a picture of microcomputer technology in the 1990s. Michael Doherty challenged his predictions (Open Forum, March 1983), requesting more substantial discussion. Grigonis' lengthy response in the August 1983 DDJ *discussed topics ranging from silicon theory to sociology. Now Grigonis has provided us with more food for thought. Doherty, having seen an advance copy of the piece, has already written a reply, which we will publish next month. – Editor*

The favorable response to my last article, "And Still More Fifth Generation Computers" (*DDJ* No. 82), has encouraged me to write another article on future hardware that picks up where the last one left off, taking speculation about future computers to the limits that can be conceived of by the human mind. So, without further ado, let us complete our journey into the future....

Physical Limitations

All artificial intelligence programs must represent knowledge and the logical connections between such knowledge, the total possible combinations of these being the search space or all possible final states or solutions. A tree structure is usually the most convenient way of describing the possible final states, with each successor node representing a possible state that can be derived from the initial state via the appropriate operators.

The search for a desired goal, such as the best move in a chess game, thus consists of searching each successor node of the tree (graph) to discover a node with a state description satisfying the goal; the path to that node is the appropriate operator sequence that yields the best final state.

Blind search or "exhaustive search" algorithms (such as the breadth-first search) never fail to uncover the best solution since they examine every node or possible state. Unfortunately, many problems have a search space that is either infinite (such as logic) or else so large (such as chess) that the computational cost makes such exhaustive searches impossible. The fact that one must examine all sequences of n moves immediately implies a search space where the number of nodes grows exponentially with n, so if at each move the player has K choices, then n moves comprise K^n possible move sequences. The possible final states grow exponentially with n and so result in a combinatorial explosion that overflows the computer's memory and requires eons of processing time. The algorithm for playing a perfect game of chess is not beyond the capabilities of a theoretical universal Turing machine, but it is certainly beyond its physical embodiment, the digital computer.

Developing artificial intelligence thus became a matter of devising short-cuts, or forms of heuristic search, so that information about the nature of the problem domain allows the program to "prune the search tree" and run more efficiently. Some of these heuristic searches include the Minimax and Alpha-Beta pruning procedures. Still, the danger with pruning the search tree via short-cuts (heuristics) is that, while some of these heuristics do guarantee success by improving efficiency, others do not even guarantee a solution at all! This is particularly troublesome when one is dealing with yes-no questions, as opposed to those questions requiring a value (with a tolerable margin of error) as an answer. The greater the number of heuristics, the more like a human being the program will behave, and the greater the chance that the program will miss the best solution. Human beings use heuristics a good deal more than they do precise algorithmic techniques, and the certainty of their decisions suffers accordingly. Only an exhaustive search of the available possibilities can give a perfect solution, but the computational cost of performing these searches is tremendous.

Therefore, many mathematical, logical, and artificial intelligence problems can never be solved because the computational requirements of algorithms that perform better than the heuristic capabilities of humans exceed the ability of any existing or projected computer. Even an optical computer utilizing matrix processing techniques will probably never exceed 1.5 trillion operations per second; this signal flow falls far short of even Hans J. Bremermann's relatively small theoretical limit of 1.35×10^{47} bits per second per gram of system weight. Searching through a search space requires computations, and computations require time and energy. Since computational devices are bound by both limits here in the physical world, as Bremermann says, "the accessible portion of the mathematical universe is limited."

Or is it? Memory is not the problem it used to be. The new Cray and Fujitsu supercomputers can each be equipped with 256 megabytes of internal memory, and the Japanese Fifth Generation computer will have 1000 megabytes or about a billion bytes (gigabyte) of memory. Sometime in the first half of the next century, supercomputers will have at their disposal about 64 gigabytes of memory. It is the *processing time* required for a solution to many of these algorithms that seems to be the real problem. In the case of chess alone, with over 10^{120} moves (I think Shannon's estimate of this is wrong, as the real figure probably exceeds 10^{150} moves), a huge number of computation-years are required to exhaustively search the possible moves.

The two solutions to this problem have been either to develop distributed array processing (reducing a problem to its components and having a different processor work on each part of the problem) or to simply build faster processors with smaller or newer non-von Neumann architectures. Neither of these ideas has turned out to be quite as successful as was initially thought. After all, even if every atom in our galaxy were replaced with a Cray-1 supercomputer, it would still take centuries to search through all the possible chess move sequences! Data flow devices have not demonstrated very many advantages over conventional computers (indeed it is impossible for them to implement simple loops or recursion), and faster and faster processors designed along the von Neumann architecture (even ignoring the problems posed by the so-called "von Neumann bottleneck") must be constructed smaller and smaller until they literally disappear.

We must conclude then that even the much-touted Japanese fifth generation computer is doomed to fail in the artificial intelligence and predicate logic theorem resolution tasks envisioned by its sponsors.

by Richard Grigonis

Richard Grigonis, Children's Television Workshop, One Lincoln Plaza, New York, NY 10023.

All drawings by Doug Kirby

Superluminal Processors?

A novel solution, which everyone seems to have missed, is to devise a processor whose signals are not bound by the speed barrier imposed by the special theory of relativity – in other words, to build a computer where the processor signals can travel faster than light. This sounds impossible, but in fact we may achieve this in three ways:

(1) Tachyons

(2) The application of the Einstein-Podolsky-Rosen (EPR) effect

(3) The so-called "advanced potentials" solution of the moving charge equations derived from Maxwell's electromagnetic theory

Let us now examine each of these three possibilities in detail and see what we can come up with.

Tachyons

Although ordinary particles with finite, real rest masses and energies cannot travel faster than electromagnetic radiation with respect to any frame of reference, particles having imaginary quantities of energy and momentum can indeed travel faster than light. These hypothetical particles were named *tachyons*, from the Greek *tachys*, meaning "swift."

In 1962 three American physicists O. Bilanuik, V. Deshpande, and E. C. G. Sudarshan at the University of Rochester mathematically demonstrated that tachyons could have *real* energy and momentum if the particles possessed an *imaginary* rest mass instead of a real one. The value for the rest mass is imaginary in that it is the square root of a negative number.

But another problem was related to the faster-than-light (or "superluminal") velocities of such hypothetical particles. Since anything traveling faster than light should, from our frame of reference, travel backwards in time as well, an observer watching a tachyon being exchanged between two atoms (say, between atom A and B) would see a tachyon particle of *negative* energy absorbed by atom B *before* it is emitted by A. Two things are wrong here. First, particles of negative energy are not allowed; second, the tachyon violates causality by traveling faster than light and so appears to reach atom B before it is emitted by atom A!

These theoretical obstacles to the existence of tachyons were eliminated by Columbia University physicist Gerald Feinberg and the "reinterpretation principle" he devised in 1967. An observer in the correct frame of reference (a high relativistic velocity) would not see a negative-energy tachyon being absorbed by atom B before it is emitted by A. Since emitting a negative amount of energy is equivalent to absorbing a positive amount, the observer instead would see *atom B emit a positive-energy tachyon*, which is then absorbed by atom A. The problems of both negative energy and time-reversal vanish.

Because they have an imaginary rest mass, as tachyons lose energy they speed up instead of slowing down! Tachyons therefore would be difficult to control. Whereas an ordinary particle is stationary when at rest and requires the application of an infinite amount of vector forces (energy) to attain the speed of light, a tachyon is subject to the opposite principle: it travels at infinite velocity when at rest, and subjecting it to an infinite amount of propulsive force would merely slow it down to the speed of light!

The speed of light, then, is a "barrier" between our universe and the universe of tachyons. Interestingly, the time-axis of the tachyons' world lines (as described by a classical Minkowski space-time diagram) is *perpendicular* to the time-axis of the particles in our own universe, so that once an observer arrives in a tachyon universe, the tachyons would appear to behave like normal particles – but the particles in our universe, by comparison, would now appear to be tachyons! That, of course, is definitely another story.

In any case, for a tachyonic computer to function, it is necessary for the tachyon signals to interact with ordinary matter at some point – namely, by triggering the processor's binary switches.

TACHYON ACCELERATING TO INFINITE VELOCITY BY RADIATING AWAY ELECTRIC CHARGE AS A CERENKOV LIGHT CONE.

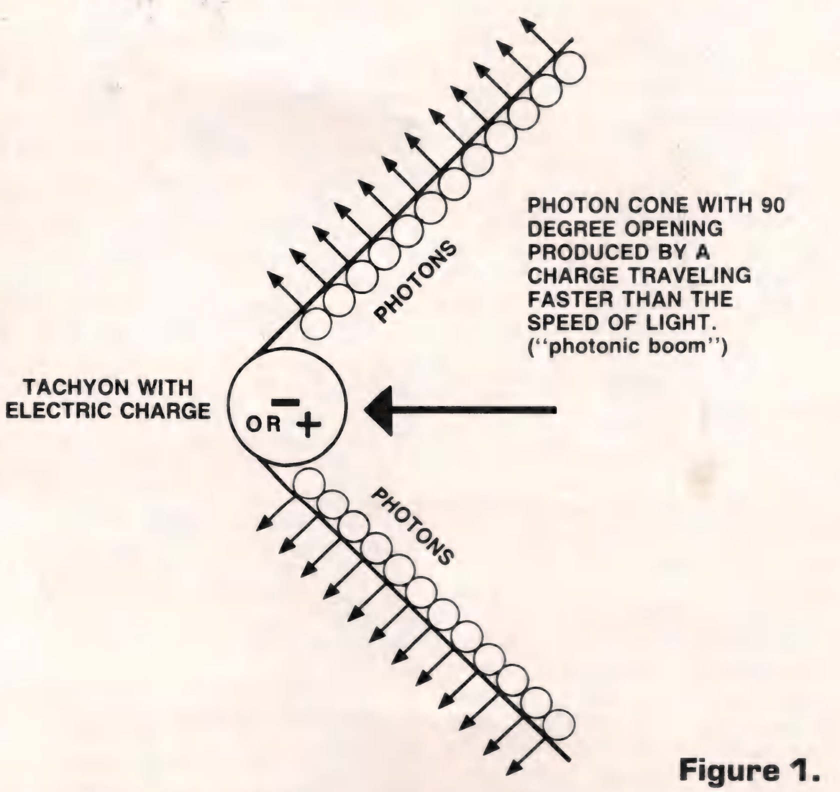


Figure 1.

If we assume that a tachyon can carry an electrical charge, we can create tachyons by bombarding atoms with high-energy photons in the form of, say, gamma rays. This then liberates pairs of tachyons, the resulting particles having equal and opposite charges. This electrical charge on the tachyon, because it is traveling at superluminal speed, emits a special form of electromagnetic radiation called Cerenkov radiation. This type of radiation, first fully described by the Russian physicist Pavel A. Cerenkov in 1937, is the cone-shaped shock wave of photons emitted by a particle traveling faster than light in a medium (not a vacuum, obviously, since ordinary particles do not travel faster than light); it is sort of a light version of the "sonic boom." (Cerenkov radiation, when generated by particles whirring around in a cyclotron, looks bluish white, but, strangely enough, a prism does not separate this light into a spectrum.) A tachyon accelerates to infinite velocity within a distance of only 0.001 centimeter, as the particle quickly divests itself of its electrical charge by liberating a Cerenkov radiation cone with a 90-degree opening.

A distance of 0.001 centimeter is too small for the construction of a super-

luminal processor switch, but a solution exists. The tachyons can be "refreshed," or given an electrical charge again, by placing an electric field in their trajectory. Once recharged in this way, the tachyons again divest themselves of their newly acquired energy via Cerenkov radiation, which can now be detected by a photomultiplier tube.

Therefore, each switch of a tachyonic computer would consist of a powerful electromagnetic source to generate the tachyons and an electrical field/photomultiplier tube combination to detect tachyonic signals coming from other switches. Since we have no way of transmitting tachyons in a particular direction, all of the switches would have to broadcast and receive signals only at certain timed intervals – in a sort of packet-switching arrangement.

Another problem is that an experiment by T. Alvager and M. N. Kriesler at Princeton in 1968, using a gamma ray source (radioactive cesium) and an electric field/light detector apparatus, did *not* detect the transmission of any tachyons! However, if high-energy tachyons merely decay into several faster and more stable tachyons, this would have accounted for the Princeton experimental results.

If tachyons are passed from one atom nucleus to another in straight lines, then disturbing one end of an object would cause a tachyonic pulse to pop out of the other end instantly. We may have here the solution to the directional transmission problem. Perhaps future computers will be partly mechanical, with the switches disturbing in some way the ends of rods, each rod composed of a highly compressed substance like metallic hydrogen (although, come to think of it, *any* substance will turn into a metal if subjected to sufficient pressure); the computer would transmit faster-than-light tachyonic pulses through a network of these rods.

Ironically, in the 1940s and 1950s, certain parts of computer processors ran so quickly in comparison to the rest of the equipment that the signals had to be slowed down at certain points by converting the signals from electrical pulses moving through wires to waves moving through troughs filled with a liquid. The Americans used mercury for this purpose, while Alan Turing thought that gin had the right density.

The EPR Effect

Einstein never liked the indeterministic, probabilistic view of the world posited by quantum mechanics. Einstein's own theories of relativity – the other half of modern physics – are actually the last of the great classical theories; their equations specify that space, time, and mass are a precise, calculable function of velocity,

and that the effect of being at rest in a gravitational field is the equivalent to being at rest in an accelerated coordinate system. Quantum physics, on the other hand, is dominated by two things that disturbed Einstein: the Schrödinger wave equation and the Heisenberg uncertainty principle.

To illustrate these two bizarre items, we reintroduce an old high school physics experiment. Most of us may remember that in 1803 Thomas Young demonstrated the wave nature of light by directing sunlight through a card or metal plate with two vertical slits in it. The diffracted light, when projected upon a wall or screen, yields an interference pattern with alternating bands of light and darkness, the brightest band at the center. With only one slit open, however, diffraction still takes place, but the light forms a single fuzzy vertical patch on the screen, corresponding to the single slit through which it has just passed.

Since not only photons but all subatomic particles have a wave aspect as well as a particle aspect, we can run the double slit experiment again, this time using electrons. Instead of an ordinary screen, however, we need one coated with a phosphor to make the electron impacts visible as tiny flashes of light (turning the apparatus into a sort of CRT); a cloud chamber would also suffice.

When a single photon is diffracted through one slit, we cannot determine exactly where on the screen the particle will land. According to the Heisenberg uncertainty principle, we can know either the momentum of a particle with accuracy or the particle's position with accuracy, but we can never determine both values with high accuracy! Since we know the momentum of a photon (traveling at light velocity), we therefore cannot determine its position – until it hits the screen.

The Schrödinger Wave Equation

But if we cannot determine the photon's precise position until it hits the screen, then can we at least determine the chances of it hitting any particular area of the screen? Yes. This is where the Schrödinger wave equation comes into play, which is simply a partial differential equation that allows one to compute, given the initial state of a system, all of the probable final states of the system. In our case, this would be all of the possible positions on the screen the photon could hit at any particular time. Instead of regarding the present state of the universe as being determined by the past and in turn determining the future (the classical view), the Schrödinger wave equation gives us a "wave function" of many possible future and past states. Thus, the future state of a physical system (until the particle finally

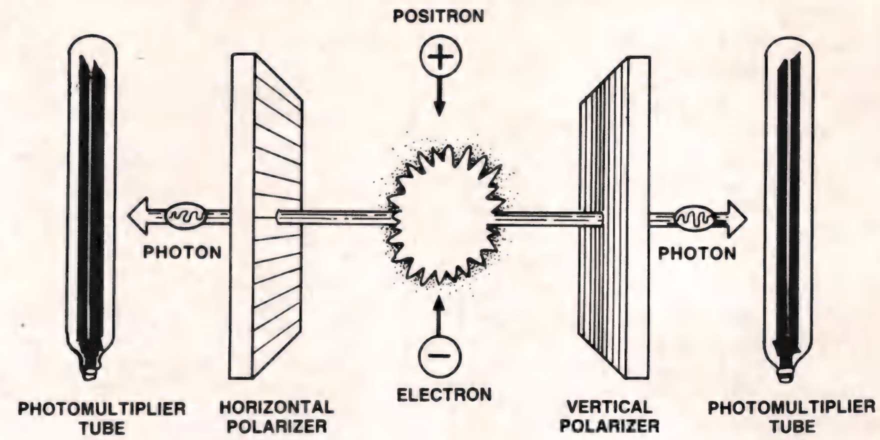
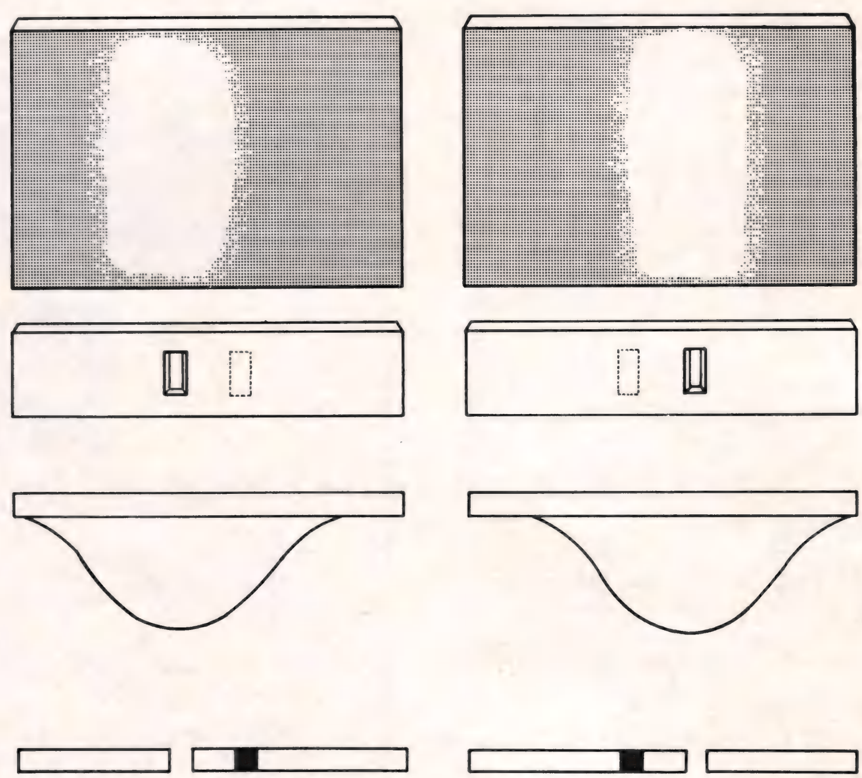


Figure 2.

EXPERIMENTAL APPARATUS TO DETECT EPR EFFECT BETWEEN TWO PHOTONS.



DOUBLE-SLIT EXPERIMENT

Figure 3.

hits) is a superposition of *all* possible outcomes.

An infinite number of states of varying probability can represent possible solutions to the Schrödinger wave equation for any physical system, depending upon the measurement. Each state is a solution to the wave equation and a possible future state (possible impact point). So, the particle can be anywhere in the cosmos before it is observed by its impact! After all, the photon or electron may have missed the screen entirely and is now far away, but this is a small probability. It is important to note that the system is not in *one* of these states, but it is probabilistically in *all* of the states until the measurement occurs; the representation of the "ensemble" of states is known as the *state vector*, or the "statistical ensemble" of states.

So imagine an electron or photon diffracted through a single slit, and imagine its wave function spread out over a huge area that includes the screen. What is so irritating about all this is that, for all practical purposes and indeed for *all* purposes, the "particle" does not exist as such until it is measured by an observer. At the moment of measurement – which is the moment when the particle hits the screen – all of the possible states disappear, the system undergoes a discontinuous change, and we find the system to be in one particular state, one given by the position on the screen of the particle's impact. The act of measurement is thus said to "collapse" the state vector onto one of the possible states given by the Schrödinger wave equation. This is also called the "reduction of the wave packet." Strangely, the particular state (position on the screen) the system settles on is determined only by pure chance.

So, although the particle is probabilistically "everywhere" (to the point where if both slits are open it can appear to pass through both slits simultaneously and interfere with itself like a wave), it is also "nowhere" in that it doesn't exist until we measure it! What is even more disturbing is that, since the wave function covers a huge area that only includes the screen, we suddenly realize that the collapse of the state vector occurs instantaneously, or much faster than the maximum speed (light velocity) allowed for by the special theory of relativity! Things like this used to drive Einstein right up the wall, leading him to say at one point that "God does not play dice with the universe."

Einstein's "Paradox" Isn't

Because Einstein could not find any inconsistency in quantum mechanics, he decided instead to prove that quantum theory was an incomplete theory. He thought he had achieved this after he cleverly concocted a "thought experiment" with two of his colleagues, Boris Podolsky and Nathan Rosen, in 1935. Unfortunately for Einstein, this particular "thought experiment" ultimately backfired, proving him wrong and quantum mechanics right. The idea is now known as the EPR (for Einstein, Podolsky, and Rosen) effect, or the EPR paradox.

There are many ways of illustrating the paradox. For example, let's bump two electrons together and create a two-particle system of zero spin, which means that the spin of one electron always cancels out the spin of the other. For the moment we will not measure the electrons to determine which way they are spinning. Next, let us separate the electrons by several light years, leaving one of them in our laboratory. After doing this we measure the spin of "our" electron, thus collapsing the state vector. Theoretically, the other electron doesn't exist until we measure it too. But we already know that both electrons make up a single two-particle system of zero spin, and the total spin is *always* going to be zero. So, even though the other electron is light years away, we already know what the value of its spin is by measuring the spin of the electron here on earth! The situation becomes even more bizarre when we remember that the other electron's spin is always going to be opposite to the spin of the one in the lab, so if we could alter the spin of the electron here on earth we would also *instantaneously* change the spin of the faraway particle!

Kind of scary, eh? Well, it's only scary if an experiment demonstrates it to be true. Einstein's argument was that such a thing would never happen, because a light signal could not travel fast enough to con-

nect the two particles. Einstein reasoned that quantum theory must be incomplete because it allows such faster-than-light (superluminal) connections between two particles, a phenomenon that must be impossible, violating as it does Einstein's own special theory of relativity.

Einstein believed in the principle of local causes – what we measure or do to one electron's spin here on earth (let us call it position S_1) cannot possibly affect the other electron's spin in deep space (let us call it position S_2) unless we allow sufficient time for some kind of signal traveling at the speed of light to reach it and impart the information to the particle. As Einstein wrote: "One can escape from this conclusion [of the incompleteness of quantum mechanics] only by either assuming that the measurement of S_1 'telepathically' changes the real situation of S_2 or by denying independent real situations as such to things which are spatially separated from each other. Both alternatives appear to me entirely unacceptable."

Amazingly, Einstein and his colleagues were proved wrong! *There are no local causes.*

Bell's Theorem

In 1964, a physicist named J. S. Bell at the European Organization for Nuclear Research (CERN) in Switzerland developed a mathematical proof that was subsequently strengthened by others and is now known as Bell's Theorem. With it, Bell theoretically demolished the principle of local causes. In 1969, J. F. Clauser, M. A. Horner, Abner Shimony, and R. A. Holt showed how physical experiments could be performed to test for faster-than-light (or "superluminal") connections or "correlations" between particles. The first of these tests was made in 1972 by Clauser with S. J. Freedman.

The scientists electrically excited neon atoms so that, when the atoms' electrons fell back to a lower, more stable energy state, they emitted pairs of photons moving in opposite directions. The wave motion of photons can be plane polarized (vertical or horizontal) or circularly polarized (clockwise or counterclockwise), so that the plane of the wave displacement rotates as they move through space. Since these photons have a common origin, they can be considered as a two-particle system. If one of the two photons is vertically polarized, the other one is also vertically polarized. If one photon is horizontally polarized, then so is the other one. Therefore, if we measure the polarization state of one photon, we also know the value of the polarization state of the other photon, even though we have not measured it and it is far enough away so that the "information" must be traveling faster than light.

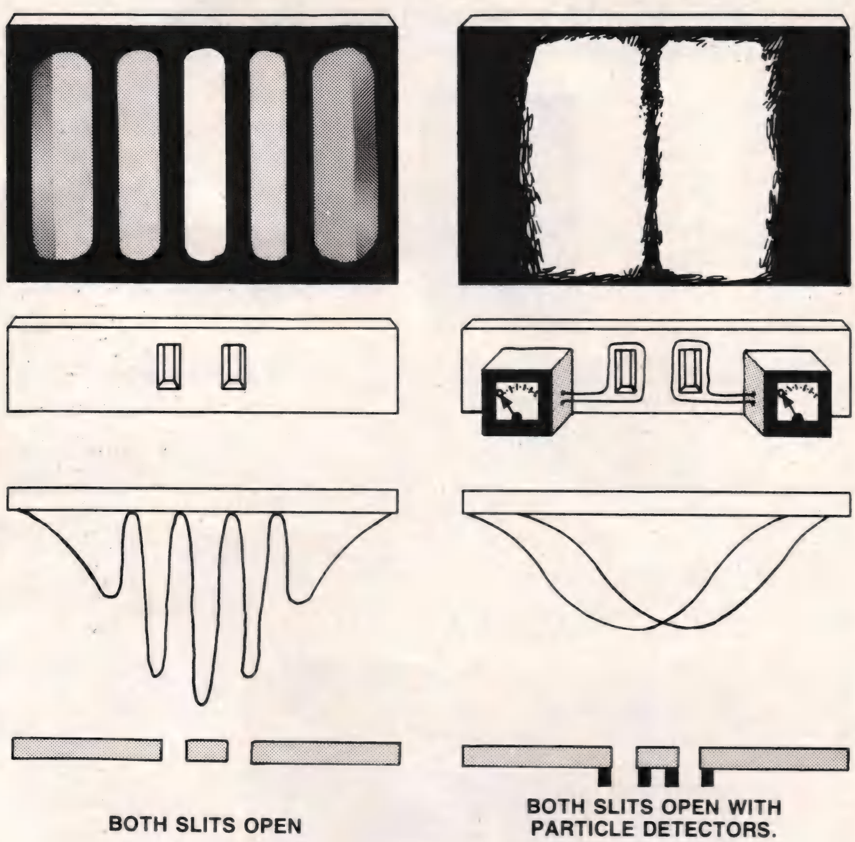
The Clauser-Freedman experiment did just that by placing polarized filters on either side of the light source. Photons passing through these polarizers then encountered photomultiplier tubes. When both polarizers were vertically oriented, the vertically vibrating photons passed through and set off both photomultipliers simultaneously. When one of the polarizers was rotated at 90 degrees to the other, however, one set of photons was stopped and the photomultipliers did not fire together.

Another way of generating pairs of photons is to bring an electron and a positron (antimatter electron) into contact, causing their annihilation. In the case of photons produced by matter-antimatter annihilation, quantum mechanics predicts that the two photons will have opposite polarization. In this case, then, the photomultipliers will fire together only if both polarizing sheets are set 90 degrees to each other. And it works again!

The most recent experiment along these lines (the eighth since 1972, I believe) was performed by Alain Aspect, Philippe Grangier, and Gerard Roger at the University of Paris. These experiments seem to prove once and for all that there are no local causes. This implies that a system can never be described in isolation, since all of the particles of matter in the universe interacted with each other long ago during the Big Bang and must therefore be "correlated" or connected.

Transmission Problems

Does all of this mean that we can build optical computers that operate faster than the speed of light? By changing an individual electron's spin or a photon's polarization, it would be quite simple to transmit messages faster than light. Quantum mechanics, however, discourages manipulating individual particles once they are measured; this violates the essential indeterminism upon which quantum theory is based. Nevertheless, while it is true that an individual photon's polarization cannot be altered once it is measured, it is also true that we decide what we are going to be looking for in the first place. Although quantum mechanics forbids individual particles to be causally tampered with, we could manipulate the variable values of a statistically large number of such particles, monitored at whatever



DOUBLE-SLIT EXPERIMENT(cont.)

Figure 4.

remote locations we wish. What immediately comes to mind is separating and altering the spin of a stream of electrons with Stern-Gerlach devices, as suggested by Gary Zukav.

Other Superluminal Phenomena

Actually, we have already encountered many kinds of faster-than-light phenomena in the double-slit experiment. First, by simply measuring the system, we have seen the wave function instantly collapse at faster than the speed of light. However, another strange phenomenon occurs that the average high school student misses when working with the apparatus. The moment an observer covers up one of the slits, preventing one set of photons or electrons from passing through it, the interference pattern consisting of light and dark bands disappears and is replaced by the single fuzzy band indicative of photons or electrons diffracted through a single slit. Now, at the moment that the observer covers up the slit, how do the other particles that are passing through the other slit "know" that there is now only one slit and that it is now permissible to go into what would have been the dark bands on the screen were both slits open, thus giving us the single fuzzy band characteristic of a single slit?

The problem is accentuated further when we fire one photon or electron at a time through the slits, slowly generating the single fuzzy band or the interference pattern on the screen, depending upon whether one or two slits are open. In this case the photon or electron not only knows what the other particles are doing, but it also knows what the physical configuration of the apparatus is! How does a single particle know if one or two slits are open?

Indeed, things become even more disturbing when one realizes that, since the spacing between the light and dark bands on the screen is determined by the distance between the slits, the particle's behavior also depends upon how far it was from a slit it didn't go through! Does the particle actually (because of its probabilistic wavelike properties) "divide" into two parts and "wave" through the two slits, interfering with itself?

To find out, let's run the experiment again with electrons, but this time try to outwit each particle and pin down exactly which slit (or slits) it goes through by building a particle detector. This is done by placing a loop of wire around each slit. A particle in motion carrying an electrical charge (the electron) generates a magnetic field; when these lines of force cut through the loop of wire (through a photon exchange between electron and wire), it generates an electrical pulse as the particle passes through the slit. We can record this pulse. By firing one electron at a time at the apparatus, we find that each time an electron hits the phosphor-coated screen, only one particle detector will register – never both. By measuring the electron at the slit we have eliminated its probabilistic location in space and have collapsed the state vector to a particular position, which is the particular slit it has passed through. But there is more! With both slits open in this experiment, the interference pattern of multiple bands has vanished. Instead we find a pattern of *two* vertical fuzzy bands, side by side, each one resembling the pattern produced by a single slit.

What happened? Consider the uncertainty principle. By measuring the position of the electron immediately (at the slit), we force the particle to interact with the apparatus by sending a photon (the exchange particle of electromagnetism) to the loop of wire. But when a photon interacts with an electron, its momentum (mass times velocity, Planck's constant divided by the wavelength) changes by an unknown amount. This is the essence of the uncertainty principle and the price we pay for knowing the electron's position. Even worse, the value for the momentum of each electron is totally different from that of any other electron, as their positions are detected in turn.

Because we have destroyed the electron's probabilistic location in space by measuring it at a particular position (one slit or the other), the electron no longer behaves as a wave and so cannot "wave" through both slits. Instead it behaves more like a particle, the interference pattern of many light and dark bands disappears, and we see the patterns of two "single" slits side-by-side on the screen. When we do not know which slit each electron passes through, the interference pattern reappears! Sneaky particles, eh?

Interestingly, if we use an "unreliable" detector, one that merely tells us that there is a probability of, say, 0.7 that an electron went through a particular slit, we will find that the interference pattern only partly disappears – the multiple bands are made a bit fuzzy – and upon this is superimposed, also a bit fuzzy, the two-single-slit pattern!

Backward Causation?

One could resolve all of these strange phenomena (and eliminate the idea of an actual signal traveling faster than light) by assuming that the "direction of causality" might be violated when we measure something. In other words, instead of regarding the initial state of the system as deter-

mining the outcome of measurements made on it, we might regard the outcome of the measurements as determining the initial state!

As long ago as 1947, Costa de Beauregard thought that the quantum information about electron spin or photon polarization (of the EPR paradox) travels backward in time from the measured particle to the event that produced the two particles, then forward in time to the other particle. The information thus appears at the second particle just as we measure it at the first particle!

More recently, John A. Wheeler, the brilliant physicist who coined the term "black hole" some years ago, has devised a thought experiment (called the "delayed-choice, double-slit experiment") that suggests that quantum mechanics seems to allow – indeed to demand – that backward causation is real. Essentially, it consists of the same version of the experiment that used the "electron detector," except this time we are working with photons. A timed photon goes through the plate with the two slits. Beyond the two slits are two different types of optical measuring equipment.

One piece of equipment can measure the photon as a *wave*. It doesn't determine which slit the photon goes through, thereby allowing the photon to move probabilistically as a wave through both slits and to interfere with itself; this "causes" the photon to strike only those areas of the screen corresponding to the bright bands of the interference pattern (areas of constructive interference), thus triggering this wave detector. The other device can measure the photon as a *particle*, by detecting through which slit the photon passes.

Nothing appears new in this experiment thus far; it just sounds like the previous experiment with the electrons and the loops of wire. But before continuing, it should be noted that, although photons (and other particles) appear to have the properties of both particles and waves, they never display both properties simultaneously. When we are measuring position, we find a particle, and when we are measuring momentum, we find a wave. We cannot measure for both properties at the same time.

Now, says Wheeler, because we can't use both pieces of detection equipment together, we have to decide whether we want to measure the photon as a wave or as a particle. But according to quantum mechanics, says Wheeler, we can decide whether the photon is a particle (goes through one slit) or a wave (probabilistically travels through both slits) *after* the photon has already gone through the plate having the slits! Particles are correlated in space *and* time. Not only do our measurements reach across space faster than light and "determine" the state of particles that we have not yet measured, but they can reach backwards in time to "justify" the present results! As Wheeler has said, "... a choice in the present can alter in an irretrievable way what we are entitled to say about the past."

Advanced Potentials

Our discussion of superluminal phenomena signalling backwards in time should actually come as no surprise; nothing in relativity theory, quantum mechanics, electrodynamics, or mechanics says that time must move in one direction. Still, we humans seem to perceive "time's arrow" in certain "obviously irreversible" processes, such as wave motion asymmetry: an electromagnetic wave (such as a radio wave) expands from the transmitting source into infinity, never to return. This typical electromagnetic phenomenon is mathematically described in a solution to the moving charge equations (derived from James Clerk Maxwell's well-known electromagnetic field equations) as $(t + r/v)$, where t is time, r is the distance of the field from a moving charge, and v is the phase velocity. This is known as the *retarded* potential solution. The resultant type of wave motion is called retarded wave motion, since the electromagnetic fluctuation can be detected at a point in space distant from the origin after a period of time.

But as is the case with homogeneous differential equations (like the one P.A.M. Dirac discovered that defined the existence of both electrons *and* their time-reversed counterparts, positrons), there is another solution to the field equation: $(t - r/v)$. This is known as the *advanced* potential solution, resulting in advanced wave motion. Whereas retarded wave motion is when a wave travels from the origin outward to infinity, advanced wave motion is when a wave travels from infinity and collapses into the origin. This implies that reverse causality is real and that the wave can be detected at a remote location before it has been generated at the source!

Strangely, although this runs counter to common sense, Maxwell's equations and the laws of propagation do not favor ordinary retarded potential solutions over the more bizarre, time-reversed, advanced potential solutions. Indeed, N. Anderson, in his book *The Electromagnetic Field* (New York: Plenum, 1968), writes that, "Advanced potentials are now receiving a great deal of attention since they seem to be a means of avoiding some of the difficulties which beset electromagnetic theory. Advanced potentials were first invoked to try to solve the problem of obtaining an equation of motion for an electron moving in an electromagnetic

TRANSMISSION AND ABSORPTION OF SPHERICAL ELECTROMAGNETIC WAVES FROM A HERTZ DIPOLE EARTHED ON ONE SIDE.

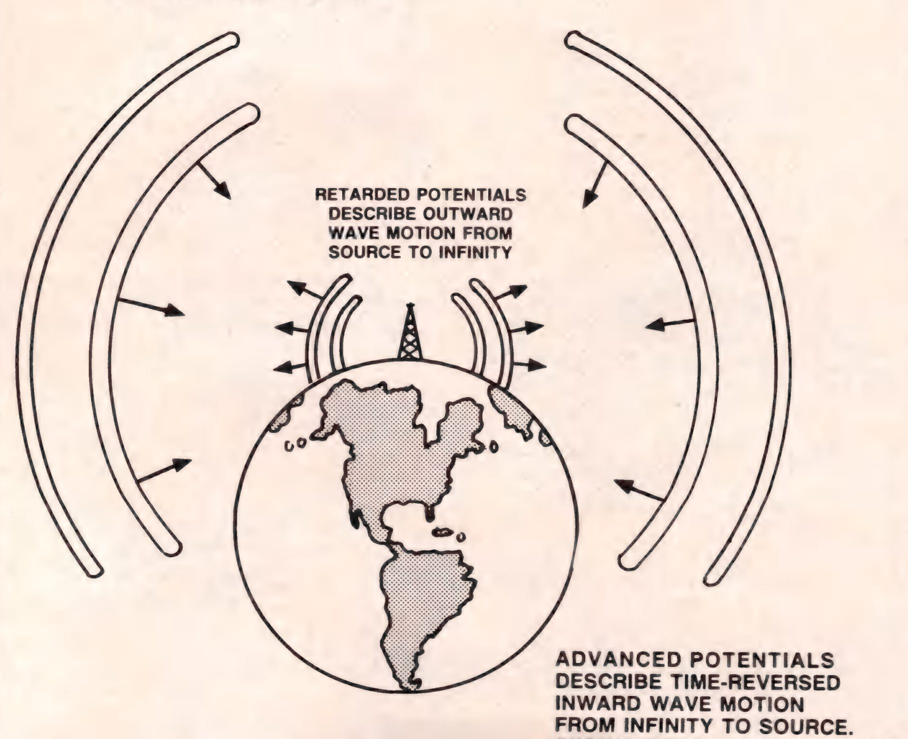


Figure 5.

field, which would take into consideration the radiation reaction, which is the force which acts on an electron due to its own electromagnetic field. Attempts using retarded potentials only are unsuccessful...."

Normally, the advanced solution is discarded, simply because no one has ever observed various parts of the distant universe "conspiring" to transmit incoming waves to radio and TV transmitters here on earth. But why doesn't this phenomenon occur? After all, it is known that various parts of the universe could be correlated in such a manner, since quantum mechanics and the EPR paradox state that all particles in the universe *do* somehow possess superluminal connections.

The best explanation centers on the fact that the universe is expanding rather than contracting or remaining stationary, so the electromagnetic radiation emanated by distant points of the universe is so red-shifted that it never reaches us. The expanding universe would even explain why time appears to travel in one direction: sources of energy can only be transmitters, never absorbers. Thus the apparent time asymmetry of the second law of thermodynamics and the "irreversible processes" that we observe associated with it are also explained. Systems can never return to their initial energy states (let alone *higher* energy states) because the universe is expanding and, as entropy increases, energy must be "diluted" more and more. An open container of liquid will evaporate away, never to return, and waves are only detectable when radiating from their sources since they never reach the other end of the universe – it keeps on expanding!

Man-Machine Communication

This should all be rather exciting to those who might one day build a computer based upon oddities in quantum theory and/or electrodynamics, allowing it to transcend the physical world in terms of processing speed. The "practical" barrier of about 1.5 trillion operations per second will be broken, but such superluminal computers will cause users additional problems.

For example, with so many signals spending a portion of their existence traveling faster than light, the large number of "negative cycles" of the processor clock that are necessary for the solution of increasingly difficult problems will result in the final answers appearing on the CRT screen or printer farther and farther back in time. At some point answers will begin appearing before the user has had a chance to key the question in!

To illustrate, let's say I wanted to ask a natural language processing AI program the question, "What prime number is closest to a trillion?" If I sat down at a terminal and keyed in this question, the answer would appear on the screen just *before* I sat down. A little disconcerting, but tolerable. Note that even though I already have the answer I (or somebody else) must at some point key in the question; otherwise, the question is never given to the processor and I will not have the answer! Sounds paradoxical, doesn't it? But it is perfectly logical. Causation is still causation, whether it be backward or forward.

But let's say that the answer appeared, and I felt particularly lazy that day and decided that I was not under any circumstances going to key in that question! What would happen? Would the universe come to an end? Would Dr. Who materialize and give me a strong lecture on space-time misbehavior?

No. When one thinks of the process only in terms of cause and effect, momentarily forgetting the direction of time, one realizes that the appearance of an answer on the screen means that a question *is* keyed into the keyboard, regardless of "time's arrow." Yours Truly or someone else *must* key in the question in the future, because now the answer is on the screen here in the past. This does not affect the idea of free will; in the future we are perfectly free to choose whether or not we want to key in the question (for example, we could decide by flipping a coin). But if we don't ask a question, of course, we

won't receive an answer in the past.

The longer it takes to work on a problem, the farther back in time the computer will answer. A truly difficult problem would take so many cycles of "negative time" to process that the answer would appear long before the question had even been formulated in the user's mind!

Now let's say that I asked the computer whether Gödel's proof is valid under all possible higher-order logics. The solution to that problem would take a great deal of negative time for the processor to come up with an answer. Perhaps years ago I saw the answer on the screen but dismissed it as a system malfunction, since I had not yet formulated the question in my mind or even knew what Gödel's proof was. Perhaps the negative time required for computing the answer is longer than the time the computer has been operational, so one never gets to see the answer! From the superluminal processor's "reversed-time" point of view, the computer would appear to have been disassembled while in the process of determining the answer. To our forward-time view, however, the computer had simply been assembled after the answer would have appeared on the screen, which would itself not have been manufactured yet. The first person to plug in the new computer, therefore, would be shocked to find the machine already working on a problem that it could never solve, since the system was not operational far enough back in time.

These problems could be alleviated – and new problems created – by typing the question into a memory buffer, just like a batch system, where a conventional computer and AI program would analyze the parameters of the question and estimate the appropriate amount of time to wait before sending the question to the superluminal processor. If the estimate were accurate, the delay period of submitting the question to the superluminal processor would be just a little longer than the negative time required for the answer. So if one keyed in a question, the answer would appear immediately afterward, even though the question had just been given to the delay buffer and not yet to the superluminal processor.

Of course, the more difficult the question, the longer it has to be kept in the delay memory buffer before being forwarded to the superluminal processor. If the user does not get an immediate response, then it means that the computer *will* be experiencing some mechanical difficulty between the time the question is keyed in and the time it is finally given to the superluminal processor. A mechanical problem or power failure is more damaging to a superluminal computer than it is to a conventional computer; although a physical interruption is the same whether one is looking at it while moving forward or backward in time, the superluminal computer user must wait for the failure to occur in order to fix it. Preventive maintenance would come in handy at this point! Normally, the user would have to wait until the mechanical problem actually occurs before keying in the question. The computer is thus not only a diagnostician of its own mechanical problems but a *precognosticator* of its future mechanical problems: the cessation of the computer's activity that prevents the answer's appearance represents another case of reverse causality, or information traveling backwards in time.

Brain-Processor Communication

With the sixth generation computer, the ultimate barrier between man and machine will finally be broken, eliminating the keyboard in favor of direct communication between the human brain and the computer. Experiments along this line have already been done, funded in some cases by CIA-type organizations in their attempts to develop a practical "thought-scanner."

When a human brain is exposed to some stimulus (such as a flash of light or the appearance of an alphanumeric symbol or an entire word), the segment of brain waves recorded by an electroencephalogram (EEG) about half a second later contains a waveform (the "evoked potential" or "event-related potential") that is a mental representation of the stimulus. The actual waveform is hidden, however, by "noise" – the unique spurious fluctuations introduced to the waveform by the individual brain undergoing analysis. The noise is eliminated and the actual waveform extracted by repeating the stimulus 50 or 100 times and then averaging all of the resulting recorded segments (each one slightly different from all the others because of the noise) and subjecting them to a Fourier analysis, which demonstrates the mathematical relationship between a complex periodic phenomenon and the simple harmonics that make it up.

The noise appears as harmonics of such complexity that for practical purposes it can be considered random noise; it cancels out once 100 samples are averaged. The final classification of the waveform includes not only the actual waveform with its fundamental wave and harmonics but the "latency" or length of time between the stimulus and the waveform's appearance.

Early success came in 1964, when W. Grey Walter and his colleagues at the Burden Neurological Institute in Bristol, England, discovered a wave that occurs whenever a subject anticipates something pleasurable (the so-called "expectancy wave"). In the same year, Samuel Sutton of the New York State Psychiatric Institute discovered the wave that we all produce when we hear a phone or doorbell or see a sudden flash of light – the so-called "surprise wave." Later, Helen Neville of the Salk Institute for Biological Studies in La Jolla, California, found the wave given off whenever one focuses one's attention on one stimulus out of many – the "selective attention wave."

In 1980, Steven Hillyard and Marta Kutas of the University of California, San Diego, discovered a negative polarity wave with a latency of 400 milliseconds (N400 wave) that represents a confused reaction to something – the "double-take wave." Thus, a person's learning process can be monitored simply by noting when during a programmed instruction course the person's brain emits the N400 wave representing his or her confusion.

In terms of identifying the language functions, neurophysiologist Donald York and speech pathologist Thomas Jensen at the University of Missouri have found "motor template waves" associated with about 20 different syllables, and a Russian scientist has supposedly isolated specific waves for specific meanings. His theory is that the different waveforms representing, for example, *chair*, *desk*, and *table* are special variations of an underlying waveform representing the concept of *furniture*, which itself could be a particular variation of a still more general waveform representing the concept of *object* or *thing*.

Practical applications of computer brain wave analysis have already appeared. Erich Sutter at the Smith-Kettlewell Institute in San Francisco has designed a system where the user dons some electrodes and glances at a particular flashing light when the user wants the computer to command a servomechanism. Each light, flashing in a pattern unique from all other lights, is delegated to specific tasks. The distinctive waveform resulting from the particular flashing-light stimulus is extracted from the brain wave pattern by a computer and is compared to the library of waveforms in the computer's memory through another pattern-matching technique. When a match occurs, the program "knows" which light was observed by the user, and a corresponding I/O subroutine is triggered to activate a certain piece of equipment. This system can help the handicapped operate bionic extremities or enable pilots to maneuver aircraft while subjected to paralyzing g-forces.

All of this work is remarkable when one realizes that less than 10 percent of the brain cells move electrical charges of sufficient strength to generate electromagnetic waves and that the average EEG records only about 32 channels. The electrical firings of the other neurons remain unanalyzed, although Robert Thatcher of the University of Maryland has described a scheme whereby a set of microwave transmitters would surround the head of an individual and the inter-

ference patterns produced between these beams and the brain's own waves would be recorded, processed by the computer, and played back as a 3-D moving picture of brain states and, therefore, mental processes – the perfect thought scanner.

Indeed, Dr. Glen Cartwright, a computer scientist and educational psychologist at McGill University, has already dubbed future brain-computer interactive devices "symbionic minds" (*sym*, as in *symbiosis*, and *bionic*).

By wearing a specially designed helmet, the sixth generation computer user will be able to locate and communicate with any person or piece of data simply by thinking about it. But why should the user be limited to a particular location, wearing a helmet that is plugged into a computer? Why not totally remove all the burdens of physical instrumentation from the user by having the computer read his brain waves remotely, without anything being worn by the user at all!

After all, the longer the wavelength of an electromagnetic wave, the farther it will travel before losing its attenuation. As it turns out, the brain produces waves of a remarkably low frequency. These are the delta waves (1 to 2 cycles per second), theta waves (3 to 6 cycles per second), alpha waves (7 to 14 cycles per second), and beta waves (15 to 30 cycles per second).

A 5 cycle-per-second electromagnetic wave (such as a brain wave) loses only 5 percent of its total energy after traveling 10,000 kilometers. If a human cranium radiates a millionth of a watt at 5 cycles per second, one would measure about 10^{-24} watts per square centimeter on the other side of the world. This doesn't sound like much until one realizes that radio astronomers have detected remnant background radio noise from the Big Bang, emanating from deep in space, which is only about 10^{-40} watts per square centimeter.

Additionally, experiments in global communications with submarines indicate that some of these frequencies set up resonances; the timing is such that a wave, weakened from its round-the-world journey, is reinforced upon its return by the next broadcast wave. The lowest of these frequencies are 7.8, 14.1, and 20.3 cycles per second, two of these falling within the alpha region and one in the beta region, the realm of ordinary waking consciousness!

Conversely, electromagnetic waves can be broadcast from the computer to the human brain to complete the communication. Physicist John Taylor has remarked that, "An aggregate of a thousand million nerve cells, each a millimeter in length, could act as a folded aerial with a total length of over a thousand kilometers." Muscle and nerve reactions to electromagnetic radiation between 50 and 200 cycles per second have reportedly been discovered in humans.

This kind of brain-computer communication would require huge antennas at the computer's location in order to receive the brain waves. We may, therefore, take one last step and apply the EPR effect of quantum mechanics to the problem. Instead of communicating by electromagnetic waves, the most advanced versions of the sixth generation computer (appearing sometime around the middle of the next century) will allow direct EPR coupling between the electrons in the human brain and the electrons in the computer's "transmitter/receiver."

In the 1951 edition of David Bohm's *Quantum Theory*, he comments on the idea of quantum aspects of brain processes that was put forth in an earlier work by Bohr (*Atomic Theory and the Description of Nature*): "In addition to such a classically describable mechanism that seems to act like a general system of communications [within the brain], Bohr's suggestion involves the idea that certain key points controlling this mechanism (which are, in turn, affected by the actions of this mechanism) are so sensitive and delicately balanced that they must be described in an essentially quantum-mechanical way. (We might, for example, imagine that such key points exist at certain types of nerve junctions.)"

One could now turn all of this around and ask if a sixth generation computer would, in fact, be conscious, since its "transmitter/receiver" would consist of halves of multiple two-particle systems correlated with their twin particles (and hence the mental processes) in the user's brain. This may be a meaningless question, since we do not even have a hint of what consciousness is in the brain.

Still, one is almost forced to speculate: are the laws of thought, even emotions and appreciations of beauty, reducible to physics, to special limiting cases of quantum theory? The pertinent facts could be distressing to those who would like to preserve a unique aspect of themselves despite the lessons of Copernicus, Darwin, and Freud, but fortunately for myself, I have just run out of editorial space.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 194.**

Bibliography

Anderson, N. *The Electromagnetic Field*. New York: Plenum, 1968.

Ashby, R. "Some Consequences of Bremermann's Limit for Information Processing Systems," in *Cybernetic Problems in Bionics* (Bionics Symposium, 1966). Edited by H. L. Oestreicher and D. R. Moore, New York: Gordon & Breach, 1968.

Bell, J. S. "On the Problem of Hidden Variables in Quantum Mechanics," *Review of Modern Physics*, 38 (1966). An elaboration of some ideas contained in his original paper as found in: *Physics*, 1 (1964-5) 195.

Birkhoff, Garrett; Von Neumann, John. "The Logic of Quantum Mechanics," *Annals of Mathematics*, 37 (1936).

Bledsoe, W. W. "A Basic Limitation of the Speed of Digital Computers," *IRE Trans. Electr. Comp.*, EC-10 (1961) 530.

Bohm, David. *Quantum Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1951.

Born, Max. (Ed.) *The Born-Einstein Letters*. New York: Walker, 1971. See pp. 164-65, 168-76, 178, 188-89, 214-15.

Bremermann, H. J. "Part I: Limitations on Data Processing Arising from Quantum Theory," in "Optimization Through Evolution and Recombination," in *Self-Organizing Systems*. Edited by M. C. Yovits, G. T. Jacobi and G. D. Goldstein, Washington, D. C.: Spartan Books, 1962.

Bremermann, H. J. "Quantum Noise and Information," in *Proc. Fifth Berkeley Sympos. Math. Stat. a. Prob.*, Berkeley, CA: University of California Press, 1967.

Bremermann, H. J. "Complexity of Automata, Brains and Behavior," in *Physics and Mathematics of the Nervous System.* Edited by M. Conrad, W. Güttinger and M. Dal Cin; *Biomathematics Lecture Notes*, Vol. 4, Heidelberg: Springer Verlag, 1974.

Clauser, J. F.; Horne, M. A.; Shimony, A.; Holt, R. A. "Proposed Experiment to Test Hidden-Variable Theories," *Physics Review* (Letters) 23 (1969).

Clauser, John F.; Shimony, Abner. "Bell's Theorem: Experimental Tests and Implications," *Reports on Progress In Physics*, 41 (1978) 1881-1927.

Costa de Beauregard, O. *Revue Internationale de Philosophie*, 1: no. 61-62 (1962), and his subsequent paper found in *Dialectica*, 19 (1965) 280.

Davies, P. C. W. *The Physics of Time Asymmetry*. London: Surrey University Press, 1974.

Einstein, Albert. "Autobiographical Notes," in *Albert Einstein, Philosopher-Scientist*. Edited by Paul Schilpp, New York: Harper and Row, 1949.

Einstein, Albert; Podolsky, Boris; Rosen, Nathan. "Can Quantum-Mechanical Description of Reality Be Considered Complete?" *Physical Review*, 47 (1935) 777-780.

Freedman, Stuart J.; Clauser, John F. "Experimental Test of Local Hidden Variable Theories," *Physical Review Letters*, 28 (1972) 938ff.

Hall, Joseph; Kim, Christopher; McElroy, Brien; Shimony, Abner. "Wave-Packet Reduction as a Medium of Communication," *Foundations of Physics*, 7 (October 1977) 759-767.

Heisenberg, Werner. *Physics and Beyond.* New York: Harper and Row, 1971.

Hooker, Clifford A. "Concerning Einstein's, Podolsky's, and Rosen's Objection to Quantum Theory," *American Journal of Physics*, 38 (July 1970) 851-857.

Hooker, Clifford A. "The Nature of Quantum Mechanical Reality: Einstein Versus Bohr," In *Paradigms and Paradoxes*. Edited by Robert G. Colodny, Pittsburgh: University of Pittsburgh Press, 1972.

Knuth, D. E. "Mathematics and Computer Science: Coping with Finiteness," *Science*, 194 (1976) 1235-1242.

Kolata, Gina. "Does Gödel's Theorem Matter to Mathematics?" *Science*, 218 (1982) 779-780.

Popper, Karl. *The Logic of Scientific Discovery*. New York: Basic Books, 1959. See pp. 244-45, 444-48, 457-64.

Present, Gerald. "Understanding How the Universe Came Into Being: An Interview with John Archibald Wheeler," *Science Digest*, 86 (October 1975) 38-45.

Reichenbach, H. *The Direction of Time*. Berkeley: University of California Press, 1971.

Schrödinger, Erwin. "Discussions of Probability Relations between Separated Systems," *Proceedings of the Cambridge Philosophical Society*, 31 (1935) 555-562.

Selden, Gary. "Machines that Read Minds," *Science Digest*, 89 (October 1981) 60ff.

Stapp, Henry P. "The Copenhagen Interpretation and the Nature of Space-Time," *American Journal of Physics*, 40 (1972) 1098.

Stapp, Henry P. "Bell's Theorem and World Process," *Il Nuovo Cimento*, 29B (1975) 271.

Stapp, Henry P. "Are Superluminal Connections Necessary?" *Il Nuovo Cimento*, 40B (1977) 191-205.

Stratton, J. A. *Electromagnetic Theory*. New York: McGraw-Hill, 1941.

Taylor, John G. *The New Physics*. New York: Basic Books, 1972.

Taylor, John G. *Superminds*. New York: Viking, 1975.

Von Neumann, John. *The Mathematical Foundations of Quantum Mechanics*. (trans. Robert T. Beyer) Princeton: Princeton University Press, 1955.

Zukav, Gary. *The Dancing Wu Li Masters: An Overview of the New Physics.* New York: William Morrow, 1979.

A New Library for Small-C

The last installment on Small-C (*DDJ* Nos. 74 and 75) presented a meager function library that failed to stress compatibility with the Unix libraries. That shortcoming was quickly pointed out, and a number of people have gone on to develop their own "standard" libraries.

This article describes one such library, which was developed jointly by the authors. It was implemented under CP/M 2.2 and provides full support for the Small-C compiler and those programs that it compiles. Virtually all of the Unix functions that apply to foreign environments have been included. Naturally, standard files with I/O redirection and Unix-style command-line argument passing are supported. The MACRO-80 package was chosen for the project.

Other than for the arithmetic and logic library (a load module in this library), which remains essentially as Ron Cain presented it in 1980, only about 20 lines of assembly language code exist in this implementation. That makes even the low-level system functions much easier to understand and to adapt to other environments.

Some changes were made to the compiler itself with this implementation, so it has been redesignated **Version 2.1**. Except for the library, however, the differences from Version 2 are minor, so we only mention them in passing:

(1) To reduce command-line clutter, a filename in the command line (not a redirection specification) causes the compiler to output to a file having the same name but with an extension of MAC; the default input file extension is C. If more than one file is specified, they are compiled into a single program bearing the name of the first file. If no filename is given, input and output are done on the standard input and output files, as before, and redirection may be used to change the default console assignments.

(2) Undeclared functions are automatically declared to be external.

(3) The syntax (*func) ( ) for declaring pointers to functions and for calling such functions is now accepted.

(4) To accommodate the MACRO-80 package, the code-generating logic was changed and some functions were renamed to avoid clashes with reserved symbols and because MACRO-80 limits external names to six characters.

(5) Fixes and enhancements reported in *DDJ* by Andrew Macpherson and Paul West have been applied, along with a number of other minor fixes.

(6) Calls to nonstandard functions have been replaced by standard library calls.

A book entitled *The Small-C Handbook* is being printed by the Reston Publishing Company and should be available about the time you read this. It fully documents, from the user's point of view, the Small-C language and compiler, including this library. Most of the material in this article's section "User Functions" is borrowed from the book.

by James Hendrix and Ernest Payne

James Hendrix, Box 8378, University, MS 38677-8378.
Ernest Payne, 1331 W. Whispering Hills Dr., Tucson, AZ 85704.

Library Organization

Generally, each library function is compiled and assembled separately; it is then kept in a library of relocatable object modules, which we call CLIB.REL. Some functions that share common code or are otherwise related are grouped into a single module. Examples are printf, fprintf, and the low-level system functions found in the module CSYSLIB.

At link time, L80 is directed to search CLIB.REL to resolve external references. Everything needed to support the program under CP/M is loaded and linked into the resulting COM file. Modules that are not referenced are not loaded. The minimum set of functions loaded comes to 5.5K bytes.

Since L80 does not scan backwards to find a module, the library is organized so that backward references only involve modules that are known to be loaded; otherwise, the library is arranged alphabetically. The compiler always generates an external reference to _link:

```
_link: : ext _main
         end
```

This occurs first in the library and forces the loading of CSYSLIB, which follows. The last module in the library is CALL, the arithmetic and logic library. It is loaded last in order to establish the location where free memory begins.

System Functions

The low-level system functions in the source file CSYSLIB.C are shown in Listing One (page 60). The names of these functions and the global variables that they use begin with the underscore character to avoid clashes with user-written function and variable names. MACRO-80 accepts these despite a statement to the contrary in the documentation. We found, however, that older versions of MACRO-80 would not accept such names as external references.

Program Initiation and Termination

The last part of CALL contains the following code:

```
_end:  lhld  6            ;get bdos address
       sphl               ;use for base of stack
       lxi   h,_end       ;get start of free memory
       shld  _memptr##    ;use for memory allocation
       jmp   _main##      ;parse command line,
                            ;execute program
       end   _end
```

The label _end designates the end of the program and the beginning of free memory. As indicated by the last line, this is also where program execution begins. (L80 plants a jump to this address at the beginning of the user program.) This logic is executed once, after which its memory space becomes available for use by the program. It first sets SP to the base of BDOS, thus overlaying the CCP; then sets _memptr to the beginning of free memory; and finally jumps to _main to prepare for execution of the program.

The function _parse is called by _main to perform command-line parsing and I/O redirection. It first copies the CP/M

command line into a dynamically allocated buffer and then scans it, calling _field to isolate arguments and _redirect to alter the assignments of stdin and stdout (appending is supported). If a redirection open fails, the program aborts after displaying the letter R for "redirection error."

Finally, argc and argv are pushed onto the stack, and main is called to start program execution. On return, exit is called with a zero argument signifying successful completion. Of course, the program could also call exit directly and pass whatever error code it wishes; the error code, if it is nonzero, is written as a byte to the console. Any open files are closed and a warm start is performed.

The BDOS Interface

A bare-bones BDOS interface is provided as a function called _bdos. It takes two arguments: first the function code to be placed in the C register, then the value to be placed in the DE register pair before calling address 5. On return, HL (the primary register for Small-C) receives the CP/M return code from the A register.

This simple interface is sufficient to support the library functions. A more complete BDOS interface was described by Terje Bolstad in *DDJ* No. 80 (June 1983). His ideas could be applied here to provide more flexibility if that is desirable.

Memory Management

Memory is allocated in unlinked, contiguous blocks beginning at the end of the program. Each call to _alloc allocates one block of zeroed or uninitialized memory, depending on the value of the second argument. The standard functions malloc and calloc call _alloc.

Memory may be deallocated by calling free or cfree, but care must be taken to deallocate memory in the reverse order from which it was allocated. Deallocating memory simply places a new value in _memptr; everything above that address is considered free.

A function called avail may be called to find out how much memory lies between _memptr and the stack pointer. If there is a program/stack overlap, avail either returns zero or aborts the program after displaying the letter M for "memory error," as requested. Malloc and calloc abort this way if sufficient memory is unavailable.

File Management

Low-level Unix functions identify files by means of small integer values called file descriptors. Whereas the Unix Standard I/O Library uses a pointer to a file control structure, our library uses the file descriptor approach throughout, even though the library includes functions from the Standard I/O Library. The impact of this difference is negligible if one restricts file references to the values returned by the function fopen and the symbols stdin, stdout, and stderr (defined in the header file STDIO.H as 0, 1, and 2, respectively).

The symbol MAXFILES in CLIB.DEF determines how many files may be opened simultaneously. Seven integer arrays are dimensioned according to that value. They are:

_status[MAXFILES] – This is a bit-encoded status word indicating whether a file is open on the corresponding file descriptor; zero implies a closed condition. Separate bits authorize reading and writing, and there are bits for end-of-file and error conditions.

_device[MAXFILES] – This contains a nonzero code designating one of the CP/M logic devices when a nondisk file is opened on the corresponding file descriptor.

_fcbptr[MAXFILES] – When a disk file is first opened on a file descriptor, a standard CP/M file control block (FCB) is dynamically allocated; its address is kept here. When the file is closed, the FCB is saved for reuse. Recall

that the memory allocation scheme is too primitive to allow indiscriminate freeing of memory. Likewise, the program must not free memory that was allocated before a disk file is opened.

_bufptr[MAXFILES] – As with the FCB, a buffer is allocated when a disk file is opened; its address is kept here. Buffers also are saved when a file is closed.

_chrpos[MAXFILES] – This is an offset to the next byte to be obtained from the corresponding buffer. Note that the current sector of a file is maintained in the random record number field in the FCB itself, and only random CP/M reads and writes are performed.

_dirty[MAXFILES] – A nonzero value here indicates that the corresponding buffer contains new data that needs to be written to disk.

_nextc[MAXFILES] – A character that has been pushed back into a file by ungetc is kept here. The value EOF (defined in STDIO.H) cannot be pushed back, so it indicates an empty bucket.

No attempt is made to save space in these arrays, since Small-C generates significantly more code to access character values than to access integers. Since these integer arrays are small, more space would be tied up in code than would be saved by making some of them character arrays.

The function _mode is used extensively to verify both that a file descriptor has a legal value and that the indicated file is open. If the file descriptor is valid, the status of its file is returned; otherwise, zero is returned.

The function _open is called by both fopen and freopen. It tries to open a file on a designated file descriptor. It verifies that the first character of the mode argument is either **r** (read), **w** (write), or **a** (append). If the filename is CON:, LST:, PUN:, or RDR:, it simply assigns the indicated logical device to the file descriptor and returns. Otherwise, it allocates an FCB and a buffer for the file, as necessary. It then calls _newfcb to validate the filename, force it to upper case, and initialize the FCB. Finally, it calls _bdos to open the file.

In the case of read mode, the first sector (128-byte CP/M record) of the file is automatically read into the buffer. In the case of write mode, if the file already exists, it is deleted before a new one is created. In the case of append mode, if the file does not exist, a new one is created; if the file does exist, it is opened, positioned at the beginning of the last block, then read to end-of-file by repeatedly calling fgetc. If a control-Z signals end-of-file, _chrpos is adjusted to begin writing at that position. Note that, while this approach avoids reading the entire file, a character stream file is presumed and embedded control-Z characters may be missed.

If a + follows the mode character (e.g., r+), an update mode is implied and _status is set to allow both reading and writing. This new feature of Unix/C is documented by C. D. Perez in *A Guide to the C Library for UNIX Users*. Apparently, under Unix/C one must call fseek or rewind when switching between read and write operations. Our library, however, permits unrestricted switching between reads and writes; each operation begins with the byte following the last one transferred. Since Small-C does not yet support long integers, we do not support fseek. Instead, cseek provides a seek to CP/M record boundaries.

When a read detects end-of-file, the EOF bit in _status is set, thereby disabling further reads; writes, however, are permitted. The EOF bit is cleared by a successful seek or rewind operation. Opening a file in write or append mode automatically sets the EOF bit. One may extend a file either by opening in append mode or by opening in read-update mode, reading to end-of-file, and then writing.

Unix performs only binary file transfers, and the end of a file is maintained as a pointer in the directory structure. It is up to the device drivers to translate between the newline character and the carriage return, line feed sequence. This scheme, however, cannot be followed under CP/M. First, there is no place in the CP/M file directory to store an end-of-file pointer. Second, in order to maintain ASCII file compatibility with other CP/M software, control-Z must be used to signal end-of-file, and the newline character must be translated to a carriage return, line feed sequence on output and vice versa on input.

Therefore, it was necessary to choose a means of discriminating between byte stream (binary) and character stream (ASCII) operations. It would not have violated the intent of the C developers to specify different, nonUnix open modes for this purpose. But we preferred to retain the standard open modes and distinguish between the I/O functions instead. In our library, calls to read, fread, write, and fwrite give binary transfers; all other calls (e.g., fgetc, fgets, etc.) give ASCII transfers. This makes Small-C programs upwardly compatible with Unix without changing the open modes.

Note that binary reads detect end-of-file only at the end of the last sector in a file. This is necessarily inconsistent with Unix, which can tie the end of a file to the byte.

Diagrammed in Figure 1 (page 54) are the principal functions involved in I/O transfers. Lines connecting the function names illustrate the possible flow of control. All input/output requests pass through either _read or _write. These perform only binary data transfers, one byte at a time. The logic for character stream operations is in fgetc and fputc, which in turn are called by the other character stream functions.

The functions _conin and _conout perform console communication. CP/M direct console I/O is used for all console communication. On output, this gives full control of the console device to the program. It also allows the program to poll the console (by means of the function poll) for operator input while writing data to the console.

Had conventional console I/O been used, CP/M would also poll for control characters, making it a matter of chance who would get an input character. Using CP/M direct console I/O meant that the customary keyboard input services (echo, rubout, etc.) had to be built into _conin and fgets. But, as a look at these functions will show, the cost was modest.

As their names imply, _getsec and _putsec transfer a sector of data between a buffer and the disk. They are called when buffers become empty or full.

Getting a sector involves first flushing the buffer to disk if it contains new data. Next, the random record number (RRN) of the FCB is advanced by calling _advance. Finally, the data is transferred by calling _sector (which calls _bdos). The end-of-file status is set if the attempt fails.

Writing a sector differs from reading by more than the direction of the data flow. The order of calls to _sector and _advance are reversed since the RRN now describes the position in the file where the newly filled buffer should go. After transferring the data, _newbuf is called to pad the buffer with control-Z characters in anticipation of further _write calls.

In keeping with the Unix concept of treating directories as ordinary files, CSYSLIB makes disk directories look like ASCII files of filenames (one to a line) to programs that read them. A file specifier consisting of only the drive identifier (e.g., B:) is taken to mean the directory on the indicated drive. The value X: indicates the default drive. Such "filenames" may be used for redirecting the standard input file (e.g., <B:) and they are accepted by the functions fopen and freopen. This feature is a compile option controlled by the symbol DIR in CSYSLIB; it takes an additional 0.3K bytes of memory. That seems a small price to pay for the flexibility it gives. Combined with a respectable set of Small-C "software tools," this feature makes the dynamic creation of SUBMIT files for performing multi-file operations a routine affair.

User Functions*

Listing Two (to be printed next month) shows the source code for the user-level functions. Most of them are patterned after Unix counterparts. Some, however, are unique to this library; these are designated as Small-C functions. A number of symbols defined in STDIO.H are mentioned. They are:

```
#define stdin    0   /* fd for standard input file */
#define stdout   1   /* fd for standard output file */
#define stderr   2   /* fd for standard error file */
#define ERR     -2   /* error condition return value */
#define EOF     -1   /* end-of-file return value */
#define NULL     0   /* value of a null character */
```

In addition, the abbreviation fd refers to a file descriptor.

Input/Output Functions

- **fopen (name, mode) char *name, *mode;**

This function attempts to open the file indicated by the null-terminated character string at name. Mode points to a string indicating the use for which the file is to be opened. The values for mode are:

"r" – read
"w" – write
"a" – append

Read mode opens an existing file for input. Write mode either creates a new file or opens an old file and truncates it to permit writing to start at the beginning of the file. Append mode allows writing to begin at the end of an existing file or at the beginning of a new one.

In addition, there are modes that allow file updating (both reading and writing). They are:

"r+" – update read
"w+" – update write
"a+" – update append

These modes are the same as their nonupdate counterparts in terms of their effect at open time, but they also allow switching between read and write modes by interleaving calls to input and output functions.

Unless the program performs a seek or rewind operation, the next read or write operation begins at the point where the previous one finished. If the attempt to open a file is successful, fopen returns an fd value for the open file; otherwise, it returns NULL. The returned fd is then used in subsequent input/output function calls to identify the file. Only the standard files may be used without first calling fopen.

- **freopen (name, mode, fd) char *name, *mode; int fd;**

This function closes the previously opened file indicated by fd and opens a new one whose name is in the null-terminated character string at name. As for fopen, mode points to a character string indicating the open mode. It returns the original value of fd if the attempt is a success or NULL upon failure either to close the old file or to open the new one. Note, however, since the fd for the standard input file is zero, there is no way of distinguishing success from failure in that case.

- **fclose (fd) int fd;**

This function closes the specified file. If any new data is being held in the file's buffer, this data is first written to disk. It returns NULL on success or a nonzero value on error.

- **fgetc (fd) int fd;** (alias getc)

This function returns the next character from the file indicated by fd. If no more characters remain in the file or an error condition occurs, it returns EOF. The end of the file is detected when the implementation standard end-of-file character occurs or the physical end of the file is reached.

- **ungetc (c, fd) char c; int fd;**

This function logically (not physically) pushes the character c back into the file indicated by fd. The next read from that file will retrieve that character first. Only one character at a time may be held in waiting. This function returns the character itself on success; it returns EOF if a previously pushed character is being held or if c has the value of EOF (you cannot push EOF into a file). Performing a seek or rewind operation on a file causes a pushed character to be forgotten.

* *Most of the material in this section is reprinted with permission of the publisher and the author from: James E. Hendrix,* The Small-C Handbook, *Reston Publishing Co., © 1984. Reston Publishing Co., 11480 Sunset Hills Road, Reston, VA 22090.*

- **getchar ( )**

  This function is equivalent to fgetc (stdin).

- **fgets (str, sz, fd) char *str; int sz, fd;**

  This function reads up to sz-1 characters into memory from the file indicated by fd, starting at the address indicated by str. Input is terminated after transferring a newline character, and a null character is appended after the newline or in the last position if newline is not found. Fgets returns str for success; otherwise, it returns NULL for end-of-file or an error.

- **fread (ptr, sz, cnt, fd) char *ptr; int sz, cnt, fd;**

  This function reads into memory from the file indicated by fd cnt items of data, sz bytes in length, starting at the address indicated by ptr. A count of the actual number of items read is returned to the caller (this might be less than cnt if the end of the file was encountered). This function performs a binary transfer; it does not convert carriage return, line feed sequences into newline characters, and it has no special regard for end-of-file bytes. It recognizes only the physical end of the file. You should call feof to determine when the data is exhausted and ferror to detect error conditions.

- **read (fd, ptr, cnt) int fd, cnt; char *ptr;**

  This function reads cnt bytes of data into memory from the file indicated by fd, starting at the address indicated by ptr. A count of the actual number of bytes read is returned to the caller. This might be less than cnt if the end of the file was encountered. This function performs a binary transfer; it does not convert carriage return, line feed sequences into newline characters, and it has no special regard for end-of-file bytes. It recognizes only the physical end of the file. You should call feof to determine for sure when the data is exhausted and ferror to detect error conditions.

- **gets (str) char *str;**

  This function reads characters into memory from stdin, starting at the address indicated by str. Input is terminated when a newline character is encountered, but the newline itself is not transferred; a null character terminates the input string. Gets returns str for success and NULL for end-of-file or an error. Since this function may transfer any amount of data, you must check the size of the input string to verify that it has not gone beyond its allotted space.

- **feof (fd) int fd;**

  This function returns a nonzero value if the file designated by fd has reached its end. Otherwise, it returns NULL.

- **ferror (fd) int fd;**

  This function returns a nonzero value if the file designated by fd has encountered an error condition since it was opened. Otherwise, it returns NULL.

- **clearerr (fd) int fd;**

  This function clears the error status for the file indicated by fd.

- **fputc (c, fd) char c; int fd;** (alias putc)

  This function writes the character c to the file indicated by fd. It returns the character itself on success; otherwise, it returns EOF. If c is a newline character, then a carriage return, line feed pair is written.

- **putchar (c) char c;**

  This function is equivalent to fputc (c, stdout).

- **fputs (str, fd) char *str; int fd;**

  This function writes characters beginning at the address indicated by str to the file indicated by fd. Successive characters are written until a null byte is found. The null byte is not written and a newline character is not appended.

- **puts (str) char *str;**

  This function works like fputs (str, stdout) except that it appends a newline character to the output.

- **fwrite (ptr, sz, cnt, fd) char *ptr; int sz, cnt, fd;**

  This function writes from memory to the file indicated by fd cnt items of data, sz bytes long, starting at the address indicated by ptr. It returns a count of the number of items written.

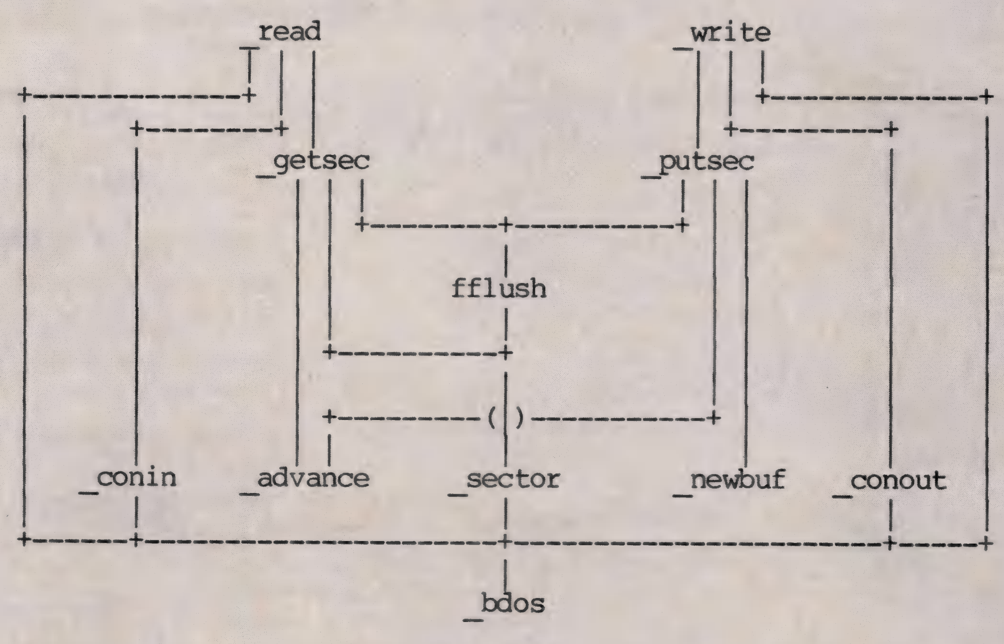


Figure 1.

Although an error condition may cause the number of items written to be less than cnt, you should call ferror to verify all error conditions. This function performs a binary transfer; it does not convert newline characters into carriage return, line feed sequences.

- **write (fd, ptr, cnt) int fd, cnt; char *ptr;**

  This function writes from memory to the file indicated by fd cnt bytes of data, starting at the address indicated by ptr. It returns a count of the number of bytes written. An error condition may cause the number of bytes written to be less than cnt. You should call ferror to verify error conditions, however. This function performs a binary transfer; it does not convert newline characters into carriage return, line feed sequences.

- **fflush (fd) int fd;**

  This function forces any system-buffered changes out to the file. Ordinarily, data written to a disk file is held in a memory buffer until the buffer becomes full, the buffer space is needed to hold a different sector of data from the disk, or the file is closed. Fclose calls this function. Fflush returns NULL on success or EOF on error.

- **cseek (fd, offset, from) int fd, offset, from;**

  This Small-C function positions the file indicated by fd to the beginning of the 128-byte record that is offset positions from the first record, current record, or end-of-file, depending on whether from is 0, 1, or 2, respectively. Subsequent reads and writes proceed from that point. It returns NULL for success and EOF otherwise.

- **rewind (fd) int fd;**

  This function positions the file indicated by fd to its beginning. It is equivalent to a seek to the first byte of the file. It returns NULL on success and EOF otherwise.

- **ctell (fd) int fd;**

  This Small-C function returns the position of the current record of the file indicated by fd. The returned value is the offset of the current 128-byte record with respect to the first record of the file. If fd is not assigned to a disk file, -1 is returned.

- **unlink (name) char *name;** (alias delete)

  This function deletes the file indicated by the null-terminated character string at name. It returns NULL on success and ERR otherwise.

- **rename (old, new) char *old, *new;**

  This Small-C function changes the name of the file specified by old to the name indicated by new. It returns NULL on success and ERR otherwise.

- **auxbuf (fd, size) int fd, size;**

  This Small-C function allocates an auxiliary buffer of size bytes for fd. It returns zero on success and ERR on failure. Fd must be open, and size must be greater than zero and less than the amount of free memory. If fd is a device, the buffer is allocated but ignored. Extra buffering is useful in reducing disk head movement or drive switching during sequential operations. Once an auxiliary buffer is allocated, it sticks for the duration of program execution, even if fd is closed. Calling this function a second time for the same fd returns ERR but otherwise has no effect. Alternating read and write operations or performing seeks produces unpredictable results; ungetc( ), however, will operate normally. Ordinarily, it is counterproductive to allocate auxiliary buffers to both input and output files.

- **iscons (fd) int fd;**

This Small-C function returns a nonzero value if fd is assigned to the console; otherwise, it returns NULL.

- **isatty (fd) int fd;**

This function returns a nonzero value if fd is assigned to a device rather than a disk file; otherwise, it returns NULL.

Formatted Input/Output Functions

- **printf(str, arg1, arg2, . . .) char *str;**

This function writes to the standard output file a formatted character string consisting of the null-terminated character array at str, laced at specific points with the character-string equivalents of the arguments:

arg1, arg2 . . .

It also returns a count of the total number of characters written.

The string at str, which is called a control string, is required, but the other arguments are optional. The control string contains ordinary characters and groups of characters called conversion specifications. Each conversion specification informs printf how to convert the corresponding argument into a character string for output. The converted argument then replaces its conversion specification in the output. The character % signals the start of a conversion specification, and one of the letters b, c, d, o, s, u, or x ends it.

Between these may be found, in the order listed and with no intervening blanks, a minus sign, a decimal integer constant, and/or a decimal fraction. These subfields are all optional; in fact, one frequently sees conversion specifications with none of them. The minus sign indicates that the string, produced by applying a specified conversion to its argument, is to be left-adjusted in its field in the output. The decimal integer indicates the minimum width of that field (in characters); if more space is needed, it will be used, but at least the indicated number of positions will be generated. The decimal fraction is used where the argument being converted is itself a character string (or more correctly, the address of a character string). In this case the decimal fraction indicates the maximum number of characters to take from the string. If the specification has no decimal fraction, then all of the string is used.

The terminating letter indicates the type of conversion to be applied to the argument. It may be one of the following:

b The argument should be considered an unsigned integer and converted to *binary* format for output. No leading zeroes are generated. This specification is unique to Small-C and should be used with that in mind.

c The argument should be output as a *character* without any conversion, in which case the high-order byte will be ignored.

d The argument should be considered a signed integer and converted to a (possibly signed) *decimal* digit string for output. No leading zeroes are generated. The sign is the leftmost character; it is blank for positive and "–" for negative.

o The argument should be considered an unsigned integer and converted to *octal* format for output. No leading zeroes are generated.

s The argument is the address of a null-terminated character *string* that should be output as is, subject to the justification, minimum width, and maximum size specifications indicated.

u The argument should be considered an unsigned integer and converted to an *unsigned* decimal character string for output. No leading zeroes are generated.

x The argument should be considered an unsigned integer and converted to *hexadecimal* format for output. No leading zeroes are generated.

If a % is followed by anything other than a valid specification, the % is ignored and the next character is written without change. So %% writes %.

Printf scans the control string from left to right, sending everything to stdout until it finds a % character. It then evaluates the conversion specification that follows and applies it to the first argument (following the control string). The resultant string is written to stdout. Printf then resumes writing data from the control string until it finds another conversion specification; it applies that one to the second argument. The procedure continues until the control string is exhausted. The result is a formatted output message consisting of both literal and variable data.

- **fprintf(fd, str, arg1, arg2, . . .) int fd; char *str;**

This function works like printf except that output goes to the file indicated by fd.

- **scanf(str, arg1, arg2, . . .) char *str;**

This function reads a series of fields from the standard input file, converts the fields to internal format according to conversion specifications contained in the control string str, and stores them at the locations indicated by the arguments:

arg1, arg2, . . .

It returns a count of the number of fields read.

A field in the input stream is a contiguous string of graphic characters. It ends with the next white-space character (blank, tab, or newline) unless its conversion specification indicates a maximum field width, in which case it ends when the field width is exhausted. A field normally begins with the first graphic character after the previous field; that is, leading white space is skipped.

Since the newline character is skipped while searching for the next field, scanf reads as many input lines as required to satisfy the number of conversion specifications in its control string. Each of the arguments following the control string must yield an address value.

The control string contains both conversion specifications and white space (which is ignored). Each conversion specification informs scanf how to convert the corresponding field into internal format, and each argument following str indicates the address where the corresponding converted field is to be stored. The character % signals the start of a conversion specification, and one of the letters b, c, d, o, s, u, or x ends it.

Between these may be found, with no intervening blanks, an asterisk and/or a decimal integer constant. As in printf, these subfields are both optional. The asterisk indicates that the corresponding field in the input stream is to be skipped; skip specifications do not have corresponding arguments. The numeric field indicates the maximum field width (in characters). If present, it causes the field to be terminated when the indicated number of characters has been scanned, even if no white space is found; however, if a white-space character is found before the field width is exhausted, the field is terminated at that point.

The terminating letter indicates the type of conversion to be applied to the field. It may be one of the following:

b The field should be considered a *binary* integer and converted to an integer value. The corresponding argument should be an integer address. Leading zeroes are ignored. This specification is unique to Small-C and should be used with that in mind.

c The field should be accepted as a single *character* without any conversion. This specification inhibits the normal skip over white-space characters. The argument for such a field should be a character address.

d The input field should be considered a (possibly signed) *decimal* integer and converted into an integer value. The corresponding argument should be an integer address.

Leading zeroes are ignored.

o The field should be considered an *octal* integer and converted to an integer value. The corresponding argument should be an integer address. Leading zeroes are ignored.

s The field should be considered a character *string* and stored with a null terminator at the character address indicated by its argument. There must be enough space at that address to hold the string and its terminator. (Remember, you can specify a maximum field width to prevent overflow.) The specification %1s will read the next graphic character, whereas %c will read the next character, whatever it is.

u The field should be considered an *unsigned* decimal integer and converted to an integer value. The corresponding argument should be an integer address. Leading zeroes are ignored. This specification is unique to Small-C and should be used with that in mind.

x The field should be considered a *hexadecimal* number and converted to an integer value. The corresponding argument should be an integer address. Leading zeroes or a leading 0x or 0X will be ignored.

Scanf scans the control string from left to right, processing input fields until the control string is exhausted or a field is found that does not match its conversion specification. If the value returned by scanf is less than the number of conversion specifications, an error has occurred or the end of the input file has been reached. EOF is returned if no fields are processed because end-of-file has been reached.

- **fscanf(fd, str, arg1, arg2, . . .) int fd; char *str;**

This function works like scanf except that the input is taken from the file indicated by fd.

Format Conversion Functions

- **atoi(str) char *str;**

This function converts the decimal number represented by the string at str to an integer and returns its value. Leading white space is skipped, and an optional sign (+ or -) may precede the leftmost digit. The first nonnumeric character terminates the conversion.

- **atoib(str, base) char *str; int base;**

This Small-C function converts the unsigned integer of base base, represented by the string at str, to an integer and returns its value. Leading white space is skipped. The first nonnumeric character terminates the conversion.

- **itoa(nbr, str) int nbr; char *str;**

This function converts the number nbr to its decimal character-string representation at str. The result is left-justified at str with a leading minus sign if nbr is negative. A null character terminates the string, which must be large enough to hold the result.

- **itoab(nbr, str, base) int nbr; char *str; int base;**

This Small-C function converts the unsigned integer nbr to its character-string representation at str in base base. The result is left-justified at str. A null character terminates the string, which must be large enough to hold the result.

- **dtoi(str, nbr) char *str; int *nbr;**

This Small-C function converts the (possibly) signed decimal number in the character string at str to an integer at nbr and returns the length of the numeric field found. The conversion stops when dtoi finds the end of the string or any illegal numeric character. When working with 16-bit integers, a leading sign and five digits at most will be used.

- **otoi(str, nbr) char *str; int *nbr;**

This Small-C function converts the octal number in the character string at str to an integer at nbr and returns the length of the octal field found. It stops when it encounters a nonoctal digit in str. When working with 16-bit integers, six digits at most will be used.

- **utoi(str, nbr) char *str; int *nbr;**

This Small-C function converts the unsigned decimal number represented by the character string at str to an integer at nbr and returns the length of the numeric field found. It stops when it encounters the end of the string or any nondecimal character. When working with 16-bit integers, five digits at most will be used.

- **xtoi(str, nbr) char *str; int *nbr;**

This Small-C function converts the hexadecimal number in the character string at str to an integer at nbr and returns the length of the hexadecimal field found. It stops when it encounters a nonhexadecimal digit in str. When working with 16-bit integers, four digits at most will be used.

- **itod(nbr, str, sz) int nbr, sz; char *str;**

This Small-C function converts nbr to a signed (if negative) character string at str. The result is right-justified and blank-filled in str. The sign and possibly the high-order digits are truncated if the destination string is too small. It returns str. Sz indicates the length of the string. If sz is greater than zero, a null byte is placed at str[sz-1]. If sz is zero, a search for the first null byte following str locates the end of the string. If sz is less than zero, all sz characters of str are used, including the last one.

- **itoo(nbr, str, sz) int nbr, sz; char *str;**

This Small-C function converts nbr to an octal character string at str. The result is right-justified and blank-filled in the destination string. High-order digits are truncated if the destination string is too small. It returns str. Sz functions the same as in itod.

- **itou(nbr, str, sz) int nbr, sz; char *str;**

This Small-C function converts nbr to an unsigned decimal character string at str. It works like itod except that the high-order bit of nbr is taken for a magnitude bit.

- **itox(nbr, str, sz) int nbr, sz; char *str;**

This Small-C function converts nbr to a hexadecimal character string at str. In all other respects, it is identical to itoo.

String-Handling Functions

- **left(str) char *str;**

This Small-C function left-adjusts the character string at str. Starting with the first nonblank character and proceeding through the null terminator, it moves the string to the address indicated by str.

- **pad (str, ch, n) char *str, ch; int n;**

This Small-C function fills the string at str with n occurrences of the character ch.

- **reverse(str) char *str;**

This function reverses the order of the characters in the null-terminated string at str.

- **strcat(dest, sour) char *dest, *sour;**

This function appends the string at sour to the end of the string at dest. The null character at the end of dest is replaced by the leading character of sour. A null character terminates the new dest string. The space reserved for dest must be large enough to hold the result. This function returns dest.

- **strncat(dest, sour, n) char *dest, *sour; int n;**

This function works like strcat except that a maximum of n characters from the source string are transferred to the destination string.

- **strcmp(str1, str2) char *str1, *str2;**

This function returns an integer less than, equal to, or greater than zero, depending on whether the string at str1 is less than, equal to, or greater than the string at str2. Character-by-character comparisons are made, starting at the left end of the strings, until a difference is found. Comparison is based on the numeric values of the characters: str2 is considered less than str1 if str2 is equal to but shorter than str1, and vice versa.

- **lexcmp(str1, str2) char *str1, *str2;**

This Small-C function works like strcmp except that a lexicographic comparison is used. For meaningful results, only characters in the ASCII character set (0–127 decimal) should appear in the strings. Alphabetics are compared in dictionary order, with upper-case letters matching their lower-case equivalents. Special characters precede the alphabetics and are themselves preceded by the control characters except DEL, which compares highest.

- **strncmp(str1, str2, n) char *str1, *str2; int n;**

This function works like strcmp except that a maximum of n characters are compared.

- **strcpy(dest, sour) char *dest, *sour;**

This function copies the string at sour to dest; dest is returned. The space at dest must be large enough to hold the string at sour.

- **strncpy(dest, sour, n) char *dest, *sour; int n;**

This function works like strcpy except that n characters are placed in the destination string regardless of the length of the source string. If the source string is too short, null padding occurs; if it is too long, it is truncated in dest. A null character follows the last character placed in the destination string.

- **strlen(str) char *str;**

This function returns a count of the number of characters in the string at str. It does not count the null character that terminates the string.

- **strchr(str, c) char *str, c;**

This function returns a pointer to the first occurrence of the character c in the string at str. It returns NULL if the character is not found. Searching ends with the first null character.

- **strrchr(str, c) char *str, c;**

This function works like strchr except that the rightmost occurrence of the character is sought.

Character Classification Functions

The following functions determine whether or not a character belongs to a designated class of characters. They return *true* (nonzero) if it does and *false* (zero) if it does not.

- **isalnum(c) char c;**

This function determines if c is alphanumeric (A–Z, a–z, or 0–9).

- **isalpha(c) char c;**

This function determines if c is alphabetic (A-Z or a-z).

- **isascii(c) char c;**

This function determines if c is an ASCII character (decimal values 0-127).

- **iscntrl(c) char c;**

This function determines if c is a control character (ASCII codes 0-31 or 127).

- **isdigit(c) char c;**

This function determines if c is a digit (0-9).

- **isgraph(c) char c;**

This function determines if c is a graphic symbol (ASCII codes 33-126).

- **islower(c) char c;**

This function determines if c is a lower-case letter (ASCII codes 97-122).

- **isprint(c) char c;**

This function determines if c is a printable character (ASCII codes 32-126). Spaces are considered printable.

- **ispunct(c) char c;**

This function determines if c is a punctuation character (all ASCII codes except control characters and alphanumeric characters).

- **isspace(c) char c;**

This function determines if c is a white-space character (ASCII SP, HT, VT, CR, LF, or FF).

- **isupper(c) char c;**

This function determines if c is an upper-case letter (ASCII codes 65-90).

- **isxdigit(c) char c;**

This function determines if c is a hexadecimal digit (0-9, A-F, or a-f).

- **lexorder(c1, c2) char c1, c2;**

This Small-C function returns an integer less than, equal to, or greater than zero depending on whether c1 is lexicographically less than, equal to, or greater than c2. For meaningful results, only characters in the ASCII character set (0-127 decimal) should be passed. Alphabetics are compared in dictionary order, with upper-case letters matching their lower-case equivalents. Special characters precede the alphabetics and are themselves preceded by the control characters except DEL, which compares highest.

Character Translation Functions

- **toascii(c) char c;**

This function returns the ASCII equivalent of c. In systems that use the ASCII character set, it merely returns c unchanged. This function makes it possible to use the properties of the ASCII code set without introducing implementation dependencies into programs.

- **tolower(c) char c;**

This function returns the lower-case equivalent of c if c is an upper-case letter; otherwise, it returns c unchanged.

- **toupper(c) char c;**

This function returns the upper-case equivalent of c if c is a lower-case letter; otherwise, it returns c unchanged.

Mathematical Functions

- **abs(nbr) int nbr;**

This function returns the absolute value of nbr.

- **sign(nbr) int nbr;**

This function returns -1, 0, or +1, depending on whether nbr is less than, equal to, or greater than zero.

Program-Control Functions

- **calloc(nbr, sz) int nbr, sz;**

This function allocates nbr*sz bytes of zeroed memory. It returns the address of the memory block on success and zero otherwise.

- **malloc(nbr) int nbr;**

This function allocates nbr bytes of uninitialized memory. It returns the address of the memory block on success and zero otherwise.

- **avail(abort) int abort;**

This Small-C function returns the number of bytes of free memory that are available between the program and the stack. It also checks to see whether the stack overlaps allocated memory; if so and if abort is not zero, the program is aborted, and the letter "S" is displayed on the console to indicate that a stack error has occurred. If abort is zero, however, avail returns zero to the caller. This function makes it possible to make full use of all available memory, but care should be taken to leave enough space for the stack to use.

- **free(addr) char *addr;** (alias cfree)

This function frees up a block of allocated memory beginning at addr. It returns addr on success and NULL otherwise. It is necessary to free memory in the reverse order from which it was allocated. Freeing memory that was allocated before opening a file should be avoided since the open function dynamically allocates buffer and FCB space. You should not assume that closing a file relinquishes its space.

- **getarg(nbr, str, sz, argc, argv)**
  char *str; int nbr, sz, argc, *argv;

This Small-C function locates the command-line argument indicated by nbr, moves it (null-terminated) to the string str of maximum size sz, and returns the length of the field obtained. Argc and argv must be the same values that are provided to the function main when the program is started. If nbr is zero, the program name is requested; if it is one, the first argument following the program name is requested; and so on. Because CP/M does not deliver the program name to a program, an asterisk is substituted in its place. If no argument corresponds to nbr, getarg puts a null byte at str and returns EOF.

- **poll(pause) int pause;**

This Small-C function polls the console for operator input. If no input is pending, zero is returned. If a character is waiting, the value of pause determines what happens. If pause is zero, the character is returned immediately. If pause is not zero and the character is a control-S, there is a pause in program execution; when the next character is entered from the keyboard, zero is returned to the caller. If the character is a control-C, program execution is terminated. All other characters are returned to the caller immediately.

- **exit(errcode) int errcode;** (alias abort)

This function closes all open files and returns to the operating system. If errcode is not zero, it is written to the console: a program that exits with a control-G (bell), for instance, would sound the console beeper.

Availability

As with Version 2, copies of this implementation of Small-C are available for $25 (add $3 for overseas postage) in 8-inch, SSSD and NorthStar CP/M formats. There is no restriction on noncommercial distribution. A package of software tools written in Small-C is also available for $35. *The Small-C Handbook* may be obtained for $14.95. Inquiries about any of these items should be addressed to James Hendrix. For people interested in other formats and adaptations, James Hendrix is willing to act as a clearing house as long as the burden does not become too great. Please include a self-addressed, stamped envelope with your inquiry.

Conclusion

No doubt the Small-C compiler and its library will continue to develop as more and more people with access to the source code take an interest in it and report their developments. Some obvious areas for improvement in the library are:

(1) A better memory allocation scheme that permits allocation and deallocation operations to be performed in any order.

(2) Adapting this implementation to other CPUs and operating systems. This library should be especially easy to port to other environments because it is written in C and the logic seems easy to follow.

(3) Improvements in efficiency. No doubt this code can be made smaller and faster in many ways. We would hope, however, that such efforts would not obscure the simplicity and transparency of the logic.

We would like to express our appreciation to Ron Cain, who started the ball rolling in 1980, and to *Dr. Dobb's* for its continued support of Small-C.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 195.**

Small-C Library (Text begins on page 50)

Listing One

```
/*
** STDIO.H -- Standard Small-C Definitions
**
** Copyright 1984  L. E. Payne and J. E. Hendrix
*/
#define stdin    0
#define stdout   1
#define stderr   2
#define ERR   (-2)
#define EOF   (-1)
#define YES      1
#define NO       0
#define NULL     0
#define CR      13
#define LF      10
```

```
#define BELL     7
#define SPACE
#define NEWLINE LF      /*23*/ /*45*/

/*
** CLIB.DEF -- Definitions for Small-C library functions.
**
** Copyright 1983  L. E. Payne and J. E. Hendrix
**
** Credits:
** 1) This library of Small-C functions was produced
**    jointly by:
**
**    Ernest Payne
**    1331 W. Whispering Hills Drive
**    Tucson, AZ  85704
**
**    and
**
**    James E. Hendrix
**    Box 8378
**    University, MS  38677-8378
**
** 2) The function _bdos() is an adaption of
**    Gene Cotton's work reported by Ron Cain (DDJ #48).
**
** 3) The functions _parse(), _field(), and _redirect()
**    are a revision of Jan-Henrik Johansson's setarg()
**    (DDJ #74), and getarg() is a modification of his
**    revision of James Hendrix' function (DDJ #75).
**
** 4) The standard C functions were obtained from
**    "A Guide to the C Library for UNIX User's
**    by C. D. Perez of Bell Laboratories.
**
*/

/*
** Definition of CP/M FCB and additional parameters
*/
#define FCBSIZE   36  /* size of file control block */
#define DRIVE      0  /* CP/M drive designator offset */
#define NAMEOFF    1  /* CP/M file name offset */
#define NAMEOFF2  16  /* CP/M 2nd file name offset */
#define NAMESIZE   8  /* CP/M file name size */
#define TYPEOFF    9  /* CP/M file type offset */
#define TYPESIZE   3  /* CP/M file type size */
#define NTSIZE    11  /* CP/M file name & type size */
#define RRNOFF    33  /* CP/M random record number offset */
#define CPMEOF    26  /* CP/M end-of-file byte */
#define BUFSIZE  128  /* size of I/O buffer */
#define MAXFILES  10  /* maximum open files */
/*
** CP/M function calls
*/
#define CLOFIL   16  /* close file */
#define DCONIO    6  /* direct console i/o */
#define DELFIL   19  /* delete file */
#define FNDFIL   17  /* find first occurrence of a file */
#define FNDNXT   18  /* find next occurrence of a file */
#define GETPOS   36  /* get number of current sector */
#define GOCPM    00  /* go to CP/M */
#define LSTOUT   05  /* list output */
```

(Continued on next page)

Small-C Library (Listing Continued, text begins on page 50)

Listing One

```
#define MAKFIL   22  /* make file */
#define OPNFIL   15  /* open file */
#define POSEND   35  /* position file to end */
#define PUNOUT   04  /* punch output */
#define RENAME   23  /* rename file */
#define RDRND    33  /* read sector randomly */
#define RDRINP   03  /* reader input */
#define SETDMA   26  /* set dma */
#define WRTRND   40  /* write sector randomly */
/*
** Device codes
*/
#define CPMCON DCONIO /* console */
#define CPMRDR RDRINP /* reader  */
#define CPMPUN PUNOUT /* punch   */
#define CPMLST LSTOUT /* list    */
/*
** File status bits
*/
#define RDBIT      1  /* open for read */
#define WRTBIT     2  /* open for write */
#define EOFBIT     4  /* eof condition */
#define ERRBIT     8  /* error condition */
/*
** ASCII characters
*/
#define ABORT    3
#define RUB      8
#define PAUSE   19
#define WIPE    24
#define DEL    127

/*
** CSYSLIB -- System-Level Library Functions
**
** Copyright 1984  L. E. Payne and J. E. Hendrix
*/

#include stdio.h
#include clib.def
#define NOCCARGC    /* no argument count passing */
#define DIR         /* compile directory option */

/*
***************** System Variables *******************
*/

int
 *_auxsz,            /* addr of _xsize[] in AUXBUF */
 *_auxef,            /* addr of _xeof[] in AUXBUF */
  _auxrd,            /* addr of _xread() in AUXBUF */
  _auxwt,            /* addr of _xwrite() in AUXBUF */
  _auxfl,            /* addr of _xflush() in AUXBUF */

  _cnt=1,            /* arg count for main */
  _vec[20],          /* arg vectors for main */

  _status[MAXFILES] = {RDBIT, WRTBIT, RDBIT|WRTBIT},
                     /* status of respective file */
  _device[MAXFILES] = {CPMCON, CPMCON, CPMCON},
                     /* non-disk device assignments */
  _nextc[MAXFILES]  = {EOF, EOF, EOF},
                     /* pigeonhole for ungetc bytes */
  _fcbptr[MAXFILES], /* FCB pointers for open files */
  _bufptr[MAXFILES], /* buffer pointers for files */
  _chrpos[MAXFILES], /* character position in buffer */
  _dirty[MAXFILES];  /* "true" if changed buffer */

char
 *_memptr,           /* pointer to free memory. */
  _arg1[]="*";       /* first arg for main */

/*
************** System-Level Functions ****************
*/

/*
** -- Process Command Line, Execute main(), and Exit to CP/M
*/
_main() {
  _parse();
  main(_cnt,_vec);
  exit(0);
  }

/*
** Parse command line and setup argc and argv.
*/
_parse() {
  char *count, *ptr;
  count = 128;  /* CP/M command buffer address */
  ptr = _alloc(count = *count&255, YES);
  strncpy(ptr, 130, count-1);
  _vec[0]=_arg1;        /* first arg = "*" */
  while (*ptr) {
    if(isspace(*ptr)) {++ptr; continue;}
    switch(*ptr) {
      case '<': ptr = _redirect(ptr, "r", stdin);
                continue;
      case '>': if(*(ptr+1) == '>')
                     ptr = _redirect(ptr+1, "a", stdout);
                else ptr = _redirect(ptr,   "w", stdout);
                continue;
      default:  if(_cnt < 20) _vec[_cnt++] = ptr;
                ptr = _field(ptr);
      }
    }
  }

/*
** Isolate next command-line field.
*/
_field(ptr) char *ptr; {
  while(*ptr) {
    if(isspace(*ptr)) {
      *ptr = NULL;
      return (++ptr);
```

(Continued on page 64)

Available now.

The Best of Both Worlds

High-performance

CompuPro Hardware

with high-performance

Cromemco Software.

Cromemco's MC68000-Z80 CROMIX multi-tasking operating system with drivers to support CompuPro hardware. (requires Cromemco DPU)

Minimum configuration:
DPU - SystemSupport1 - Disk1 - Interfacer3 or 4 - 192K RAM

CROMIX with drivers to support minimum configuration $890.
Special drivers only $295.

The following products can be added to any 8 or 8-16 bit CROMIX system:

SCSI hard disk drivers $195.
15Mb formatted hard disk subsystem $2095.
30Mb (two drives) $2995.

MDrive-H drivers $195.
CompuPro 512K MDrive-H hardware $1695.

Other drivers in development... custom inquiries welcome.

Authorized CompuPro Dealer.

PuterParts®
2004 4th Avenue
P.O. Box 12339
Seattle, WA 98111-4339
Telephone (206) 682-2590

OPEN
2-6 Tuesday-Saturday

"PuterParts" is a registered trademark of Katharos Companies
Additional trademarks: Z80, Zilog; MC68000, Motorola; CROMIX, DPU, Cromemco; SystemSupport1, Disk1, Interfacer3, -4, MDrive-H, CompuPro.

Circle **no. 56** on reader service card.

8748

CP/M SIMULATOR/DEBUGGER FOR THE INTEL 8748/8048

ANNOUNCING SIM48

- SIM48 allows you to load, trace, execute and save Intel 8748/8048 software using standard Intel Hex files.
- SIM48 allows you to simulate all instruction operations, timer/counter operations, I/O operations, interrupt processing, reset execution, internal and external RAM and ROM.
- SIM48's command set includes load, breakpoint, assemble, list (disassemble), trace, call and execute commands (as seen in DDT and ZSID).
- SIM48 allows you to simulate all software operations of the 8748/8048, yet costs 1/20th of an In-Circuit Emulator.
- SIM48 is CP/M compatible. Supplied on an 8" SSSD diskette.
- SIM22 (for Intel 8021/8022's) and SIM51 (for Intel 8751/8051's) soon to be released.

SIM48 .... $150.00 SIM48 Manual $20.00

Plus shipping and handling.
N.Y. State residents add sales tax.
Mastercard/Visa

Logical Systems
6184 TEALL STATION
SYRACUSE, NY 13217
(315) 457-9416

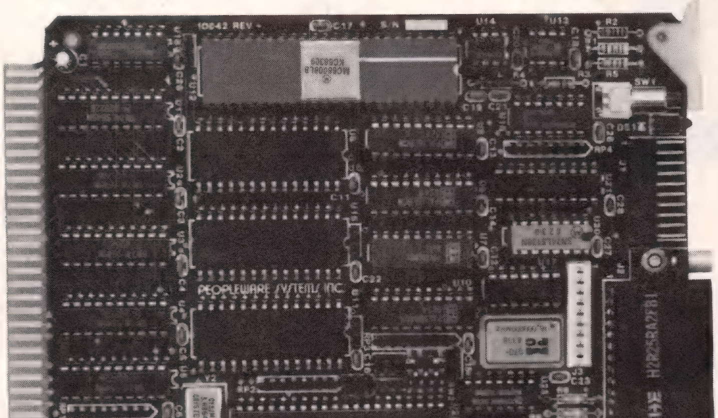
SIM48, SIM22, and SIM51 are trademarks of Logical Systems Corporation.
CP/M, ZSID, and DDT are trademarks of Digital Research.

Circle **no. 37** on reader service card.

68000 SINGLE BOARD COMPUTER FOR STD BUS

FEATURES

- 8 MHz 68008 Processor
- 32K On-board Memory Space (three 28-pin JEDEC sockets)
- One Megabyte Address Range (non-multiplexed address bus)
- 68681 DUART
- Dual RS232 Serial I/O Ports
- TTL Parallel I/O Port



FEATURES

- 16-Bit Timer
- Four Layer PC Board
- STD Bus Compatible
- Monitor EPROM Optional

ADDITIONAL PRODUCTS

- 6801 P-FORTH SBC
- I/O Cards
- Six Slot Card Cage with Integral Power Supply

ENGINEERING SERVICES ALSO AVAILABLE.

PEOPLEWARE SYSTEMS, INC.
5190 West 76th Street • Minneapolis, Minnesota 55435
612-831-0827 TWX 9105761735

CALL OR WRITE US TODAY 612-831-0827

Circle **no. 50** on reader service card.

Small-C Library (Listing Continued, text begins on page 50)

Listing One

```
      }
    ++ptr;
    }
  return (ptr);
  }

/*
** Redirect stdin or stdout.
*/
_redirect(ptr, mode, std)  char *ptr, *mode; int std; {
  char *fn;
  fn = ++ptr;
  ptr = _field(ptr);
  if(_open(fn, mode, std)==ERR) exit('R');
  return (ptr);
  }

/*
** ------------ File Open
*/

/*
** Open file on specified fd.
*/
_open(fn, mode, fd) char *fn, *mode; int fd; {
  char *fcb;
  if(!strchr("rwa", *mode)) return (ERR);
  _nextc[fd] = EOF;
  if(_auxef) _auxef[fd] = NO;
  if(strcmp(fn,"CON:")==0) {
    _device[fd]=CPMCON; _status[fd]=RDBIT|WRTBIT; return (fd);
    }
  if(strcmp(fn,"RDR:")==0) {
    _device[fd]=CPMRDR; _status[fd]=RDBIT;  return (fd);
    }
  if(strcmp(fn,"PUN:")==0) {
    _device[fd]=CPMPUN; _status[fd]=WRTBIT; return (fd);
    }
  if(strcmp(fn,"LST:")==0) {
    _device[fd]=CPMLST; _status[fd]=WRTBIT; return (fd);
    }
  if(fcb = _fcbptr[fd]) pad(fcb, NULL, FCBSIZE);
  else {
    if((fcb = _fcbptr[fd] = _alloc(FCBSIZE, YES)) == NULL
         || (_bufptr[fd] = _alloc(BUFSIZE, YES)) == NULL)
      return (ERR);
    }
  pad(_bufptr[fd], CPMEOF, BUFSIZE);
  _dirty[fd] = _device[fd] = _chrpos[fd] = 0;
#ifdef DIR
  if(fn[1] == ':' && fn[2] == NULL) {  /* directory file */
    pad(fcb, NULL, FCBSIZE);
    pad(fcb+NAMEOFF, '?', NTSIZE);
    if(toupper(fn[0]) != 'X') *fcb = toupper(fn[0]) - 64;
    _chrpos[fd] = BUFSIZE;
    _device[fd] = FNDFIL;
    _status[fd] = RDBIT;
    return (fd);
    }
```

```
#endif
  if(!_newfcb(fn,fcb)) return (ERR);
  switch(*mode) {
    case 'r': {
      if(_bdos(OPNFIL,fcb)==255) return (ERR);
      _status[fd] =  RDBIT;
      if(_sector(fd,  RDRND)) _seteof(fd);
      break;
      }
    case 'w': {
      if(_bdos(FNDFIL,fcb)!=255) _bdos(DELFIL,fcb);
    create:
      if(_bdos(MAKFIL,fcb)==255) return (ERR);
      _status[fd] = EOFBIT|WRTBIT;
      break;
      }
    default: {      /* append mode */
      if(_bdos(OPNFIL,fcb)==255) goto create;
      _status[fd] = RDBIT;
      cseek(fd, -1, 2);
      while(fgetc(fd)!=EOF) ;
      _status[fd] = EOFBIT|WRTBIT;
      }
    }
  if(*(mode+1)=='+') _status[fd] |= RDBIT|WRTBIT;
  return (fd);
  }

/*
** Create CP/M file control block from file name.
** Entry: fn  = Legal CP/M file name (null terminated)
**              May be prefixed by letter of drive.
**        fcb = Pointer to memory space for CP/M fcb.
** Returns the pointer to the fcb.
*/
_newfcb(fn, fcb) char *fn, *fcb; {
  char *fnptr;
  pad(fcb+1, SPACE, NTSIZE);
  if(*(fn + 1) == ':') {
    *fcb = toupper(*fn) - 64;
    fnptr = fn + 2;
    }
  else fnptr = fn;
  if(*fnptr == NULL) return (NO);
  fnptr = _loadfn(fcb + NAMEOFF, fnptr, NAMESIZE);
  if(*fnptr == '.') ++fnptr;
  else if(*fnptr) return (NO);
  fnptr = _loadfn(fcb + TYPEOFF, fnptr, TYPESIZE);
  if(*fnptr) return (NO);
  return (YES);
  }

/*
** Load into fcb and validate file name.
*/
_loadfn(dest, sour, max) char *dest, *sour; int max; {
  while(*sour && !strchr("<>.,;:=?*[]", *sour)) {
    if(max--) *dest++ = toupper(*sour++);
    else break;
    }
  return (sour);
  }

/*
** ------------ File Input
*/

/*
** Binary-stream input of one byte from fd.
*/
_read(fd) int fd; {
  char *bufloc;
  int ch;
  switch (_mode(fd)) {
    default: _seterr(fd); return (EOF);
    case RDBIT:
    case RDBIT|WRTBIT:
    }
  if((ch = _nextc[fd]) != EOF) {
    _nextc[fd] = EOF;
    return (ch);
    }
  switch(_device[fd]) {
    /* PUN & LST can't occur since they are write mode */
    case CPMCON: return (_conin());
    case CPMRDR: return (_bdos(RDRINP,NULL));
    default:
         if(_auxsz && _auxsz[fd]) return (_auxrd(fd));
         if(_chrpos[fd]>=BUFSIZE && !_getsec(fd))
           return (EOF);
         bufloc = _bufptr[fd] + _chrpos[fd]++;
         return (*bufloc);
    }
  }

/*
** Console character input.
*/
_conin() {
  int ch;
  while(!(ch = _bdos(DCONIO, 255))) ;
  switch(ch) {
    case ABORT: exit(0);
    case    LF:
    case    CR: _conout(LF); return (_conout(CR));
    case   DEL: ch = RUB;
       default: if(ch < 32) { _conout('^'); _conout(ch+64);}
                else _conout(ch);
                return (ch);
    }
  }

/*
** Read one sector from fd.
*/
_getsec(fd) int fd; {
#ifdef DIR
  if(_device[fd]) {        /* directory file */
    char *bp, *name, *type, *end;
    _bdos(SETDMA, 128);
    if((name = _bdos(_device[fd], _fcbptr[fd])) == 255) {
      _seteof(fd);
      return (NO);
      }
    _device[fd] = FNDNXT;
    name = (name << 5) + (128 + NAMEOFF);
    type = name + NAMESIZE;
    end = name + NTSIZE;
```

(Continued on next page)

Small-C Library (Listing Continued, text begins on page 50)

Listing One

```
    bp = _bufptr[fd] + BUFSIZE;
    *--bp = CR;
    while(--end >= name) { /* put filename at end of buffer */
      if(*end == SPACE) continue;
      *--bp = *end;
      if(end == type) *--bp = '.';
      }
    _chrpos[fd] = bp - _bufptr[fd];
    return (YES);
    }
#endif
  if(fflush(fd)) return (NO);
  _advance(fd);
  if(_sector(fd, RDRND)) {
    pad(_bufptr[fd], CPMEOF, BUFSIZE);
    _seteof(fd);
    return (NO);
    }
  return (YES);
  }

/*
** ------------ File Output
*/

/*
** Binary-Stream output of one byte to fd.
*/
_write(ch, fd) int ch, fd; {
  char *bufloc;
  switch (_mode(fd)) {
    default: _seterr(fd); return (EOF);
    case WRTBIT:
    case WRTBIT|RDBIT:
    case WRTBIT|EOFBIT:
    case WRTBIT|EOFBIT|RDBIT:
    }
  switch(_device[fd]) {
    /* RDR can't occur since it is read mode */
    case CPMCON: return (_conout(ch));
    case CPMPUN:
    case CPMLST: _bdos(_device[fd], ch);
                 break;
    default:
      if(_auxsz && _auxsz[fd]) return (_auxwt(ch, fd));
      if(_chrpos[fd]>=BUFSIZE && !_putsec(fd)) return (EOF);
      bufloc = _bufptr[fd] + _chrpos[fd]++;
      *bufloc = ch;
      _dirty[fd] = YES;
    }
  return (ch);
  }

/*
** Console character output.
*/
_conout(ch) int ch; {
  _bdos(DCONIO, ch);
  return (ch);
```

```
  }

/*
** Write one sector to fd.
*/
_putsec(fd) int fd; {
  if(fflush(fd)) return (NO);
  _advance(fd);
  pad(_bufptr[fd], CPMEOF, BUFSIZE);
  return (YES);
  }

/*
** ------------ Buffer Service
*/

/*
** Advance to next sector.
*/
_advance(fd) int fd; {
  int *rrn;
  rrn = _fcbptr[fd] + RRNOFF;
  ++(*rrn);
  _chrpos[fd] = 0;
  }

/*
** Sector I/O.
*/
_sector(fd, func) int fd, func; {
  int error;
  _bdos(SETDMA, _bufptr[fd]);
  error = _bdos(func, _fcbptr[fd]);
  _bdos(SETDMA, 128);
  _dirty[fd] = NO;
  return (error);
  }

/*
** ------------ File Status
*/

/*
** Return fd's open mode, else NULL.
*/
_mode(fd) char *fd; {
  if(fd < MAXFILES) return (_status[fd]);
  return (NULL);
  }

/*
** Set eof status for fd and
** disable future i/o unless writing is allowed.
*/
_seteof(fd) int fd; {
  _status[fd] |= EOFBIT;
  }

```

(Continued on next page)

Small-C Library (Listing Continued, text begins on page 50)

Listing One

```
/*
** Clear eof status for fd.
*/
_clreof(fd) int fd; {
  _status[fd] &= ~EOFBIT;
  }

/*
** Set error status for fd.
*/
_seterr(fd) int fd; {
  _status[fd] |= ERRBIT;
  }

/*
** ------------ Memory Allocation
*/

/*
** Allocate n bytes of (possibly zeroed) memory.
** Entry: n = Size of the items in bytes.
**    clear = "true" if clearing is desired.
** Returns the address of the allocated block of memory
** or NULL if the requested amount of space is not available.
*/
_alloc(n, clear) char *n; int clear; {
  char *oldptr;
  if(n < avail(YES)) {
    if(clear) pad(_memptr, NULL, n);
    oldptr = _memptr;
    _memptr += n;
    return (oldptr);
    }
  return (NULL);
  }

/*
** ------------ CP/M Interface
*/

/*
** Issue CP/M function and return result.
** Entry: c  = CP/M function code (register C)
**        de = CP/M parameter (register DE or E)
** Returns the CP/M return code (register A)
*/
_bdos(c,de) int c,de; {
#asm
        pop     h       ;hold return address
        pop     d       ;load CP/M function parameter
        pop     b       ;load CP/M function number
        push    b       ;restore
        push    d       ;  the
        push    h       ;    stack
        call    5       ;call bdos
        mvi     h,0     ;
        mov     l,a     ;return the CP/M response
#endasm
  }
```

End Listing One

Listing Two

File: ABS.C

```
/*
** abs -- returns absolute value of nbr
*/
abs(nbr)  int nbr; {
  if(nbr < 0) return (-nbr);
  return (nbr);
  }
```

File: ATOI.C

```
#define NOCCARGC  /* no argument count passing */
/*
** atoi(s) - convert s to integer.
*/
atoi(s) char *s; {
  int sign, n;
  while(isspace(*s)) ++s;
  sign = 1;
  switch(*s) {
    case '-': sign = -1;
    case '+': ++s;
    }
  n = 0;
  while(isdigit(*s)) n = 10 * n + *s++ - '0';
  return (sign * n);
  }
```

File: ATOIB.C

```
#define NOCCARGC  /* no argument count passing */
/*
```

(Continued on page 70)

Small-C Library (Listing Continued, text begins on page 50)

Listing Two

```
** atoib(s,b) - Convert s to "unsigned" integer in base b.
**              NOTE: This is a non-standard function.
*/
atoib(s, b) char *s; int b; {
  int n, digit;
  n = 0;
  while(isspace(*s)) ++s;
  while((digit = (127 & *s++)) >= '0') {
    if(digit >= 'a')      digit -= 87;
    else if(digit >= 'A') digit -= 55;
    else                  digit -= '0';
    if(digit >= b) break;
    n = b * n + digit;
    }
  return (n);
  }

```

File: AUXBUF.C

```
#define NOCCARGC  /* no arguement count passing */
#include stdio.h
#include clib.def
extern int *_auxsz, *_auxef, _auxrd, _auxwt, _auxfl,
            _status[];
/*
** This module is loaded with a program only if auxbuf()
** is called.  It links to _open(), _read(), _write(), and
** fflush() through _auxsz, _auxef, _auxrd, _auxwt, and _auxfl
** in CSYSLIB.  This technique reduces the overhead for
** programs which don't use auxiliary buffering.  Presumably,
** if there is enough memory for extra buffering, there is
** room to spare for this overhead too.  A bug in some
** versions of Small-C between 2.0 and 2.1 may cause the calls
** to _auxrd, _auxwt, and _auxfl in _read(), _write(), and
** fflush(), respectively, to produce bad code.  The current
** compiler corrects the problem.
*/
int
  _xsize[MAXFILES],  /* size of buffer */
  _xaddr[MAXFILES],  /* aux buffer address */
  _xnext[MAXFILES],  /* address of next byte in buffer */
  _xend[MAXFILES],   /* address of end-of-data in buffer */
  _xeof[MAXFILES];   /* true if current buffer ends file */
/*
** auxbuf -- allocate an auxiliary input buffer for fd
**   fd = file descriptor of an open file
** size = size of buffer to be allocated
** Returns NULL on success, else ERR.
** Note: Ungetc() still works.
**       A 2nd call returns ERR, but has no effect.
**       If fd is a device, buffer is allocated but ignored.
**       Buffer stays allocated when fd is closed.
**       Do not mix reads and writes or perform seeks on fd.
*/
auxbuf(fd, size) int fd; char *size; {   /* fake unsigned */
  if(!_mode(fd) || !size || avail(NO) < size   || _xsize[fd])
    return (ERR);
```

```
  _xaddr[fd] = _xnext[fd] = _xend[fd] = malloc(size);
  _auxef = _xeof;     /* tell _open() where _xeof[] is */
  _auxrd = _xread;    /* tell _read() where _xread() is */
  _auxwt = _xwrite;   /* tell _write() where _xwrite() is */
  _auxsz = _xsize;    /* tell both where _xsize[] is */
  _auxfl = _xflush;   /* tell fflush() where _xflush() is */
  _xsize[fd] = size;  /* tell _read() that fd has aux buf */
  return (NULL);
  }

/*
** Fill buffer if necessary, and return next byte.
*/
_xread(fd) int fd; {
  char *ptr;
  while(YES) {
    ptr = _xnext[fd];
    if(ptr < _xend[fd]) {++_xnext[fd]; return (*ptr);}
    if(_xeof[fd]) {_seteof(fd); return (EOF);}
    _auxsz = NULL;            /* avoid recursive loop */
    _xend[fd] = _xaddr[fd]
              + read(fd, _xnext[fd]=_xaddr[fd], _xsize[fd]);
    _auxsz = _xsize;          /* restore _auxsz */
    if(feof(fd)) {_xeof[fd] = YES; _clreof(fd);}
    }
  }

/*
** Empty buffer if necessary, and store ch in buffer.
*/
_xwrite(ch, fd) int ch, fd; {
  char *ptr;
  while(YES) {
    ptr = _xnext[fd];
    if(ptr < (_xaddr[fd] + _xsize[fd]))
      {*ptr = ch; ++_xnext[fd]; return (ch);}
    if(_xflush(fd)) return (EOF);
    }
  }

/*
** Flush aux buffer to file.
*/
_xflush(fd) int fd; {
  int i, j;
  i = _xnext[fd] - _xaddr[fd];
  _auxsz = NULL;   /* avoid recursive loop */
  j = write(fd, _xnext[fd]=_xaddr[fd], i);
  _auxsz = _xsize; /* restore _auxsz */
  if(i != j) return (EOF);
  return (NULL);
  }
```

(Continued on next page)

Small-C Library (Listing Continued, text begins on page 50)

Listing Two

File: AVAIL.C

```
#define NOCCARGC  /* no argument count passing */
extern char *_memptr;
/*
** Return the number of bytes of available memory.
** In case of a stack overflow condition, if 'abort'
** is non-zero the program aborts with an 'S' clue,
** otherwise zero is returned.
*/
avail(abort) int abort; {
  char x;
  if(&x < _memptr) {
    if(abort) exit('M');
    return (0);
    }
  return (&x - _memptr);
  }

```

File: CALLOC.C

```
#define NOCCARGC  /* no argument count passing */
#include stdio.h
/*
** Cleared-memory allocation of n items of size bytes.
** n     = Number of items to allocate space for.
** size  = Size of the items in bytes.
** Returns the address of the allocated block,
** else NULL for failure.
*/
calloc(n, size) char *n, *size; {
  return (_alloc(n*size, YES));
  }
```

File: CLEARERR.C

```
#define NOCCARGC  /* no arg count passing */
#include stdio.h
#include clib.def
extern int _status[];
/*
** Clear error status for fd.
*/
clearerr(fd) int fd; {
  if(_mode(fd)) _status[fd] &= ~ERRBIT;
```

```
}
```

File: CSEEK.C

```
#define NOCCARGC  /* no arguement count passing */
#include stdio.h
#include clib.def
extern int _fcbptr[], _chrpos[], _nextc[];
/*
** Position fd to the 128-byte record indicated by
** "offset" relative to the point indicated by "base."
**
**     BASE     OFFSET-RELATIVE-TO
**       0      first record
**       1      current record
**       2      end of file (last record + 1)
**
** Returns NULL on success, else EOF.
*/
cseek(fd, offset, base) int fd, offset, base; {
  int oldrrn, *rrn;
  if(!_mode(fd) || isatty(fd) || fflush(fd)) return (EOF);
  rrn = _fcbptr[fd] + RRNOFF;
  oldrrn = *rrn;
  switch (base) {
    case 2: _bdos(POSEND, _fcbptr[fd]);
    case 1: *rrn += offset;
            break;
    case 0: *rrn = offset;
            break;
    default: return (EOF);
    }
  if(_sector(fd,  RDRND)) {
    *rrn = oldrrn;
    return (EOF);
    }
  _chrpos[fd] = 0;
  _nextc[fd] = EOF;
  _clreof(fd);
  return (NULL);
  }
```

File: CTELL.C

```
#define NOCCARGC  /* no arg count passing */
#include stdio.h
#include clib.def
extern int _fcbptr[], _chrpos[];
/*
** Return offset to current 128-byte record.
*/
ctell(fd) int fd; {
  int *rrn;
  if(!_mode(fd) || isatty(fd)) return (-1);
  rrn=_fcbptr[fd]+RRNOFF;
  return (*rrn);
  }
/*
** Return offset to next character in current buffer.
*/
ctellc(fd) int fd; {
```

(Continued on next page)

Small-C Library
Listing Two

(Listing Continued, text begins on page 50)

```
  return (_chrpos[fd]);
  }

```

File: DTOI.C

```
#define NOCCARGC  /* no argument count passing */
#include stdio.h
/*
** dtoi -- convert signed decimal string to integer nbr
**         returns field length, else ERR on error
*/
dtoi(decstr, nbr)  char *decstr;  int *nbr;  {
  int len, s;
  if((*decstr)=='-') {s=1; ++decstr;} else s=0;
  if((len=utoi(decstr, nbr))<0) return ERR;
  if(*nbr<0) return ERR;
  if(s) {*nbr = -*nbr; return ++len;} else return len;
  }
```

File: EXIT.C

```
#define NOCCARGC  /* no argument count passing */
#include stdio.h
#include clib.def
/*
** Close all open files and exit to CP/M.
** Entry: errcode = Character to be sent to stderr.
** Returns to CP/M rather than the caller.
*/
exit(errcode) char errcode; {
  int fd;
  if(errcode) _conout(errcode);
  for(fd=0; fd < MAXFILES; fclose(fd++));
  _bdos(GOCPM,NULL);
  }
#asm
abort  equ    exit
       entry  abort
#endasm
```

File: FCLOSE.C

```
#define NOCCARGC  /* no argument count passing */
#include stdio.h
#include clib.def
/*
** Close fd
** Entry: fd = File descriptor for file to be closed.
** Returns NULL for success, otherwise ERR
*/
extern int _fcbptr[], _status[], _device[];
fclose(fd) int fd; {
  if(!_mode(fd)) return (ERR);
  if(!isatty(fd)) {
    if(fflush(fd) || _bdos(CLOFIL,_fcbptr[fd])==255)
```

```
    return (ERR);
    }
  return (_status[fd]=_device[fd]=NULL);
  }

```

File: FEOF.C

```
#define NOCCARGC  /* no argument count passing */
#include clib.def
extern int _status[];
/*
** Test for end-of-file status.
** Entry: fd = file descriptor
** Returns non-zero if fd is at eof, else zero.
*/
feof(fd) int fd; {
  return (_status[fd] & EOFBIT);
  }

```

File: FERROR.C

```
#define NOCCARGC  /* no argument count passing */
#include stdio.h
#include clib.def
extern int _dirty[], *_auxsz, _auxfl;
/*
** Write buffer for fd if it has changes.
*/
ferror(fd) int fd; {
  return (_status[fd] & ERRBIT);
  }
```

File: FFLUSH.C

```
#define NOCCARGC  /* no arg count passing */
#include stdio.h
#include clib.def
extern _status[];
/*
** Test for error status on fd.
** Entry: fd = File descriptor of pertinent file.
** Returns NULL on success, otherwise EOF.
*/
fflush(fd) int fd; {
  if(!_mode(fd)) return (ERR);
  if(_auxsz && _auxsz[fd] && _auxfl(fd)) return (ERR);
  if(!isatty(fd) && _dirty[fd] && _sector(fd, WRTRND)) {
    _seterr(fd);
    return (ERR);
    }
  return (NULL);
  }

```

(Continued on next page)

Small-C Library (Listing Continued, text begins on page 50)

Listing Two

File: FGETC.C

```
#define NOCCARGC  /* no argument count passing */
#include stdio.h
#include clib.def
extern int _chrpos[];
/*
** Character-stream input of one character from fd.
** Entry: fd = File descriptor of pertinent file.
** Returns the next character on success, else EOF.
*/
fgetc(fd) int fd; {
  int ch;
  while(1) {
    switch(ch = _read(fd)) {
      default:     return (ch);
      case CPMEOF: switch(_chrpos[fd]) {
                     default: --_chrpos[fd];
                     case 0:
                     case BUFSIZE:
                     }
                   _seteof(fd);
                   return (EOF);
      case CR:     return ('\n');
      case LF:    /* NOTE: _conin() maps LF -> CR */
      }
    }
  }
#asm
getc equ   fgetc
     entry getc
#endasm

```

File: FGETS.C

```
#define NOCCARGC  /* no arg count passing */
#include stdio.h
#include clib.def
/*
** Gets an entire string (including its newline
** terminator) or size-1 characters, whichever comes
** first. The input is terminated by a null character.
** Entry: str  = Pointer to destination buffer.
**        size = Size of the destination buffer.
**        fd   = File descriptor of pertinent file.
** Returns str on success, else NULL.
*/
fgets(str, size, fd) char *str; int size, fd; {
  return (_gets(str, size, fd, 1));
  }

/*
** Gets an entire string from stdin (excluding its newline
** terminator) or size-1 characters, whichever comes
** first. The input is terminated by a null character.
** The user buffer must be large enough to hold the data.
** Entry: str  = Pointer to destination buffer.
** Returns str on success, else NULL.
*/
gets(str) char *str; {
  return (_gets(str, 32767, stdin, 0));
  }

_gets(str, size, fd, nl) char *str; int size, fd, nl; {
  int backup;
  char *next;
  next = str;
  while(--size > 0) {
    switch (*next = fgetc(fd)) {
      case  EOF: *next = NULL;
                 if(next == str) return (NULL);
                 return (str);
      case '\n': *(next + nl) = NULL;
                 return (str);
      case  RUB: if(next > str) backup = 1; else backup = 0;
                 goto backout;
      case WIPE: backup = next - str;
        backout:
                 if(iscons(fd)) {
                   fputs("\b \b\b \b", stderr);
                   ++size;
                   while(backup--) {
                     fputs("\b \b", stderr);
                     if(*--next < 32) fputs("\b \b", stderr);
                     ++size;
                     }
                   continue;
                   }
        default: ++next;
      }
    }
  *next = NULL;
  return (str);
  }

```

File: FOPEN.C

```
#define NOCCARGC  /* no arg count passing */
#include stdio.h
#include clib.def
/*
** Open file indicated by fn.
** Entry: fn   = Null-terminated CP/M file name.
**               May be prefixed by letter of dirve.
**               May be just CON:, RDR:, PUN:, or LST:.
**        mode = "a"  - append
**               "r"  - read
**               "w"  - write
**               "a+" - append update
```

(Continued on page 78)

Small-C Library (Listing Continued, text begins on page 50)

Listing Two

```
**                 "r+" - read   update
**                 "w+" - write  update
** Returns a file descriptor on success, else NULL.
*/
fopen(fn, mode) char *fn, *mode; {
  int fd;
  fd = 0; /* skip stdin (= error return) */
  while(++fd < MAXFILES) {
    if(_mode(fd) == NULL) {
      if(_open(fn, mode, fd)!=ERR) return (fd);
      break;
      }
    }
  return (NULL);
  }

```

File: FPRINTF.C

```
#define NOCCARGC
/*
** Yes, that is correct.  Although these functions use an
** argument count, they do not call functions which need one.
*/
#include stdio.h
/*
** fprintf(fd, ctlstring, arg, arg, ...) - Formatted print.
** Operates as described by Kernighan & Ritchie.
** b, c, d, o, s, u, and x specifications are supported.
** Note: b (binary) is a non-standard extension.
*/
fprintf(argc) int argc; {
  int *nxtarg;
  nxtarg = CCARGC() + &argc;
  return(_print(*(--nxtarg), --nxtarg));
  }

/*
** printf(ctlstring, arg, arg, ...) - Formatted print.
** Operates as described by Kernighan & Ritchie.
** b, c, d, o, s, u, and x specifications are supported.
** Note: b (binary) is a non-standard extension.
*/
printf(argc) int argc; {
  return(_print(stdout, CCARGC() + &argc - 1));
  }

/*
** _print(fd, ctlstring, arg, arg, ...)
** Called by fprintf() and printf().
*/
_print(fd, nxtarg) int fd, *nxtarg; {
  int  arg, left, pad, cc, len, maxchr, width;
  char *ctl, *sptr, str[17];
  cc = 0;
  ctl = *nxtarg--;
  while(*ctl) {
    if(*ctl!='%') {fputc(*ctl++, fd); ++cc; continue;}
    else ++ctl;
    if(*ctl=='%') {fputc(*ctl++, fd); ++cc; continue;}
    if(*ctl=='-') {left = 1; ++ctl;} else left = 0;
    if(*ctl=='0') pad = '0'; else pad = ' ';
    if(isdigit(*ctl)) {
      width = atoi(ctl++);
      while(isdigit(*ctl)) ++ctl;
      }
    else width = 0;
    if(*ctl=='.') {
      maxchr = atoi(++ctl);
      while(isdigit(*ctl)) ++ctl;
      }
    else maxchr = 0;
    arg = *nxtarg--;
    sptr = str;
    switch(*ctl++) {
      case 'c': str[0] = arg; str[1] = NULL; break;
      case 's': sptr = arg;        break;
      case 'd': itoa(arg,str);     break;
      case 'b': itoab(arg,str,2);  break;
      case 'o': itoab(arg,str,8);  break;
      case 'u': itoab(arg,str,10); break;
      case 'x': itoab(arg,str,16); break;
      default:  return (cc);
      }
    len = strlen(sptr);
    if(maxchr && maxchr<len) len = maxchr;
    if(width>len) width = width - len; else width = 0;
    if(!left) while(width--) {fputc(pad,fd); ++cc;}
    while(len--) {fputc(*sptr++,fd); ++cc; }
    if(left) while(width--) {fputc(pad,fd); ++cc;}
    }
  return(cc);
  }

```

File: FPUTC.C

```
#define NOCCARGC  /* no arg count passing */
#include stdio.h
#include clib.def
extern int _status[];
/*
** Character-stream output of a character to fd.
** Entry: ch = Character to write.
**        fd = File descriptor of perinent file.
** Returns character written on success, else EOF.
*/
fputc(ch, fd) int ch, fd; {
  switch(ch) {
    case EOF:  _write(CPMEOF, fd); break;
```

(Continued on page 80)

Small-C Library (Listing Continued, text begins on page 50)

Listing Two

```
    case '\n': _write(CR, fd); _write(LF, fd); break;
    default:   _write(ch, fd);
    }
  if(_status[fd] & ERRBIT) return (EOF);
  return (ch);
  }
#asm
putc equ   fputc
     entry putc
#endasm
```

File: FPUTS.C

```
#define NOCCARGC  /* no arg count passing */
#include stdio.h
#include clib.def
/*
** Write a string to fd.
** Entry: string = Pointer to null-terminated string.
**        fd     = File descriptor of pertinent file.
*/
fputs(string,fd) char *string; int fd; {
  while(*string)
    fputc(*string++,fd) ;
  }

```

File: FREAD.C

```
#define NOCCARGC  /* no argument count passing */
#include clib.def
extern int _status[];
/*
** Item-stream read from fd.
** Entry: buf = address of target buffer
**         sz = size of items in bytes
**          n = number of items to read
**         fd = file descriptor
** Returns a count of the items actually read.
** Use feof() and ferror() to determine file status.
*/
fread(buf, sz, n, fd) char *buf; int sz, n, fd; {
  return (read(fd, buf, n*sz));
  }

/*
** Binary-stream read from fd.
** Entry:  fd = file descriptor
**        buf = address of target buffer
**          n = number of bytes to read
** Returns a count of the bytes actually read.
** Use feof() and ferror() to determine file status.
*/
read(fd, buf, n) int fd, n; char *buf; {
  char *cnt;  /* fake unsigned */
  cnt = 0;
  while(n--) {
    *buf++ = _read(fd);
```

```
    if(_status[fd] & (ERRBIT | EOFBIT)) break;
    ++cnt;
    }
  return (cnt);
  }
```

File: FREE.C

```
#define NOCCARGC  /* no argument count passing */
extern char *_memptr;
/*
** free(ptr) - Free previously allocated memory block.
** Memory must be freed in the reverse order from which
** it was allocated.
** ptr    = Value returned by calloc() or malloc().
** Returns ptr if successful or NULL otherwise.
*/
free(ptr) char *ptr; {
   return (_memptr = ptr);
   }
#asm
cfree  equ    free
       entry  cfree
#endasm
```

File: FREOPEN.C

```
#define NOCCARGC  /* no argument count passing */
#include stdio.h
/*
** Close previously opened fd and reopen it.
** Entry: fn   = Null-terminated CP/M file name.
**              May be prefixed by letter of drive.
**              May be just CON:, RDR:, PUN:, or LST:.
**        mode = "a"  - append
**               "r"  - read
**               "w"  - write
**               "a+" - append update
**               "r+" - read   update
**               "w+" - write  update
**        fd   = File descriptor of pertinent file.
** Returns the original fd on success, else NULL.
*/
freopen(fn, mode, fd) char *fn, *mode; int fd; {
  if(fclose(fd)) return (NULL);
  if(_open(fn, mode, fd)==ERR) return (NULL);
  return (fd);
  }
```

(To be Continued in June Issue)

The Accent Finder

If you have ever taken an introductory Spanish or French course, you probably know what accents are. You remember: those little marks that had to be placed on top of certain letters in certain words just to make your life bitter. French has three kinds of accents; the rules for their placement are complicated and have many exceptions. Spanish has only one accent and a single placement rule with fewer exceptions. Nonetheless, things can get complicated.

If you want to write Spanish correctly, you have three choices:

1. Learn the accent placement rule and all the exceptions (time factor: 7 days)
2. Write with a dictionary by your side and look up each word (time factor: 1 minute per word)
3. Write a program that tells the user where to put the accent and teaches the user the rules and exceptions (time factor: who cares, as long as I get to play?)

I wrote the Accent Finder program as an exercise in the use of sets in Pascal and also for the benefit of UCLA students interested in using the computer for humanities. Members of the Spanish department thoroughly tested the Accent Finder, and so far it has not produced any erroneous output. Should you find an error, please contact me for changes. The UCLA IBM4341 User's Group keeps a version of this program on the school's open access computer for any interested users.

A Brief Primer of Spanish Phonology

Believe it or not, there is an academy (in Spain, naturally) that supervises the use and development of Spanish. This institution has formulated rules for the placement of the accent and for the correct parsing of words into syllables (which is crucial for our algorithm).

Spanish orthography is almost a mapping of the phonetics (i.e., each letter is a symbol for a unique sound; you write what you hear). The phonetic inventory consists of the vowels (A, E, I, O, U) and a multitude of consonants more or less similar to English. There are also two liquids: L and R. These sounds are called liquids because of certain features (i.e., they can be held a long time, just like a vowel, and the sound-generating flow of air escapes on both sides of the tongue while it touches the palate). Vowels are strong (the set [A, E, O]) and nonstrong (the set [I, U]).

by Eddy Vasile

Eddy Vasile, 3314 Sawtelle Blvd. #28, Los Angeles, CA 90066.

The Accent Placement Rule

For the purpose of placing accents, a Spanish word is described by two criteria:

1. The position of the syllable with the pronunciation emphasis
2. The quality of the last sound of the word

To decide whether an accent is needed, we apply each of the criteria to the word in question and compare the results.

Each criterion can mark a word with a plus or a minus. If the emphasis syllable is the penultimate one, then the word gets a plus; otherwise, the word gets a minus. If the word ends with a vowel, an N, or an S, then the word gets a plus from criterion 2; otherwise, the word gets a minus. If the word has two different signs (i.e., both plus and minus), then the word gets a written accent on the emphasis vowel. If the word has two identical signs, then no accent is required.

Let's look at an example. In the word *chicharon,* the syllables are:

1 CHI ** 2 CHA ** 3 RON

The emphasis is on syllable 3, not the penultimate one. Thus, criterion 1 marks the word with a minus. The last sound of the word is N. Thus, criterion 2 marks the word with a plus. The signs for *chicharon* are mixed, so you must place a written accent on the last syllable: *chicharón.*

Now let's look at *francesa.* The syllables are:

1 FRAN ** 2 CE ** 3 SA

The emphasis is on the penultimate syllable: the word gets a plus under criterion 1. The last sound is A, a vowel: the word gets another plus under criterion 2. The signs are the same, so no accent is required.

The Accent Placement Algorithm

The accent placement problem is now to find a way of coding the word description criteria (the emphasis position and the quality of the last sound). I chose to represent the criteria by means of two Booleans: CountPlus, which is true when the emphasis is on the penultimate syllable, and EndPlus, which is true when the word ends with an N, an S, or a vowel.

The setting of EndPlus is trivial. First, define the variable vowels of type SET OF CHAR:

Vowels := ['A', 'E', 'I', 'O', 'U']

Next, check to see if the last character of the word is in the vowel set. If it is, then EndPlus is set to true.

The setting of CountPlus is more complicated. The word must first be split into syllables (we'll cover that shortly), and the position of the emphasis must then be identified.

I chose to identify the position of the emphasis by asking the user. This accomplishes two goals: the user thinks about the rule and the pronunciation, and words such as *mama,* which can take the emphasis on different syllables according to the syntactic role, are covered.

The Algorithm for Syllable Parsing

Splitting a word into syllables is a process that relies on our linguistic intuitions. However, a fairly simple set of rules can be drafted for automatic parsing. We need three arrays of Booleans parallel with the string holding the word: Strong, Vowel, and Boundary. Strong is true if the sound in the string is a strong vowel. Vowel is true if the sound is a vowel. Boundary is true if the sound at the same position in the string is the boundary between two syllables.

All we must do now is to set the values of the elements of the Boundary array. Boundary is set to true (i.e., the respective sound is a border between two syllables) in the following cases:

1. If the current sound is T and the previous sound was N
2. If the current sound is a vowel and the previous sound was a consonant
3. If the current sound is a nonstrong vowel and the previous sound was a strong vowel
4. If both the current and the previous sounds are vowels

All this is done in the SetSylls procedure via a FOR loop, which checks the qualities of each sound and its neighbor's. Each time the qualifying conditions are met, the element of the Boundary array at that position is set to true.

In the SetSylls procedure, a final check is executed. If the procedure finds

the pattern vowel-consonant-consonant, then the Boundary is reset: the first consonant gets to be a boundary, and the second becomes part of the syllable initiated by the first. If the procedure encounters two neighboring consonants, then the second one marks a syllable boundary. One exception is LL, which in Spanish is pronounced like the English Y; this is coded in the procedure Split-Consonants, which checks for a consonant-consonant pattern (other than LL) by means of a FOR loop.

Two other consonant pairs should not be split: CH, which generates a single sound, just as its English counterpart, and liquids. The procedures WeldCH and WeldLiquids scan the CH and liquid patterns (using a FOR loop) and set the Boundary to true for the leading consonants.

Now that the syllable boundaries are set, we can concatenate all the letters marked false in the Boundary array to the letter marked true that precedes them, creating the bodies of the syllables that are kept in the Syllables array.

Program Output

The program displays the syllables with a delimiter between them and asks the user to pick the one with the emphasis. Next, the plus/minus rule is applied as described above; if the word has an accent, then it is displayed on the strong vowel of the syllable with the emphasis. The program also explains its decisions as it goes along, using the procedure InfoMsg.

The program provides for extensive error checking for entry of incorrect alphanumeric symbols and incorrect syllable numbers. Special procedures also are included for words that get accents according to syntactic context.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 196.**

Accent Finder Listing

```
(**************************************************************************
The Accent program analyzes spanish words and determines if and where
there is a written accent. The program will ask for the word and then
it will display the word split in syllables and it will ask for the
number of the syllable carrying the pronounciation emphasis. The
syllables are split by stars.
If there should be a written accent, then its position will
be displayed.

The algorithm has considerable error checking with the use of sets
for  bad input or bogus syllable numbers.
                Author:        Eddy C. Vasile
                Last modified: Sept 15 83

NOTE: Please warn users that syllable parsing is sometimes
      disputed even in highly qualified linguistic circles.
      My parsing algorithm will produce a somewhat unorthodox
      syllable scheme when dealing with diphthongs, but only
      to suit the ultimate need of accent evaluation.
**************************************************************************)

program accent;
   const
      MaxChar = 50;      (*Maximum line size          *)
      MaxSyl  = 10;      (*Maximum number of syllables *)
      SylSize = 5;       (*Maximum size of a syllable  *)
      Radix   = 10;      (* base 10, used for conversions*)
   type
      CharSet       = set of char;
      WordType      = string[MaxChar];
      WordIndexType = 1..MaxChar;
   var
      Word     :WordType; (*The word to be processed*)
      Letters,            (*The alphabet            *)
      LowerCaseSet,       (*Lower case letters      *)
      Vowels,
      StrongVowels,
      Liquids :CharSet;
      Error:boolean;
```

(Continued on next page)

Accent Finder Listing (Listing Continued, text begins on page 82)

(Listing Continued, text begins on page 82)

```
(*************************************************************************
* function NUMERIC                                                       *
*   Given : 1-10 char token (InputStr)                                   *
*   Return : TRUE if it is a numeric constant.                           *
*            FALSE if it is not.                                         *
*   If any char is non-numeric, the token is non-numeric.                *
*************************************************************************)
function Numeric(InputStr : WordType): boolean;
   var
      i      : WordIndexType;        (* pos. chars of input string *)
   begin  (* Numeric *)
      Numeric := TRUE;                    (* return TRUE by default *)
      for i := 1 to length(InputStr) do          (* if any chars are *)
         if not (InputStr[i] in ['0'..'9'])  (* non-numeric, *)
            then Numeric := FALSE            (* return FALSE *)
   end;  (* Numeric *)

(*************************************************************************
* function CONVERT                                                       *
*   Given : a string of digits (Instring).                               *
*   Return : its value (integer).                                        *
*   Convert from string to numeric by multiplying each digit by the      *
*      appropriate power of the specified base (Radix).                  *
*************************************************************************)
function Convert(Instring:WordType):integer;
   var
      Temp : Integer;
      i    : WordIndexType;          (* poses chars of input string *)
(**********************************************************************
* The DigValue function returns the numeric value of a single digit*
**********************************************************************)
function DigValue (c:char):integer;
   begin
      DigValue:=ord(c)-ord('0')
   end;

   begin  (* Convert *)
      Temp:=0;                          (* initially return value = 0 *)

      (* Beginning with left-most digit, multiply its value by the *
       *   specified base and add result to previous value.        *)
      if length(InString) > 0 then
         begin
            Temp:=DigValue(InString[i]);
            for i:=2 to length(InString) do
               Temp:=Temp * Radix + DigValue(InString[i])
         end (*if*)
  end;  (* Convert *)
(*********************************************
* The condense procedure preprocesses the input*
* word by compressing it and trimming both ends*
* If your compiler does not have these        *
* you may either ignore this proc or write the *
* functions your self.                        *
*********************************************)
```

```
function Condense(Instring:WordType):WordType;
   begin
      Condense:=compress(ltrim(trim(Instring)))
   end; (*Condense*)

(*********************************************
*The upper case procedure uppercases a line of  *
*input. If your machine is ASCII then this is   *
*fine, if EBCDIC then you must change the       *
*conversion factor to 64 and add                *
*********************************************)
function Uppercase(InputStr : WordType): WordType;
var
   i              : WordIndexType;
begin
   for i := 1 to length(InputStr) do
      if  (InputStr[i] in LowerCaseSet)
      then InputStr[i] := chr(ord(InputStr[i])-32);
   Uppercase := InputStr
end; (* Uppercase *)

(*******************************************************
*procedure CheckChars checks for the validity of the   *
*input by identifying garbage characters             *
*******************************************************)
```

(Continued on next page)

Accent Finder Listing (Listing Continued, text begins on page 82)

```
procedure CheckChars(var Error:boolean; word:wordType);
   var
      i:WordIndexType;
   begin
      error:=false;
      for i:=1 to length(word) do
          if not (word[i] in Letters)
          then error:=true
   end; (*CheckChars*)

(*****************************************************
*procedure MImsg offers a message if the word is MI  *
*Special treatment for two letter words is necessary *
*due to the number of pecularities.                  *
*****************************************************)
procedure MImsg;
    begin
       writeln('Mi does not take an accent if it is used as a',
               ' possesive pronoun,');
       writeln('but it takes an accent when used as a reflexive',
               ' pronoun .');
       writeln('Example:Mi perro es feo');
       writeln('Me afeito a mi'' mismo.')
    end;


(*****************************************************
*procedure SImsg offers a message if the word is SI  *
*****************************************************)
procedure SImsg;
   begin
      writeln('The word SI has an accent when used as ''yes'',');
      writeln('or when used as a reflexive pronoun.');
      writeln('Example:! Si'', me duelen los dientes!');
      writeln('Se afeita a si'' mismo.')
   end;


(*****************************************************
*procedure ELmsg offers a message if the word is EL  *
*****************************************************)
procedure ELmsg;
   begin
      writeln('El has an accent when used as a pronoun and not');
      writeln('when used as an article.');
      writeln('E''l me dio un puno');
      writeln('!Deme el dinero!')
   end;

(**********************************************************
The prompt procedure will prompt user for a word, scan it
for invalid input and set an error parameter to true if
so. Else, the word is passed to the PROCESS procedure.
**********************************************************)
```

(Continued on page 88)

```
procedure prompt(var word:WordType; var Error:boolean);
   begin
      write('Enter the word > ');
      readln(word);
      word:=UpperCase(Condense(word))
    end; (*prompt*)


(***********************************************************
*The master procedure of the program that processes the   *
*word by dividing it in syllables and computing whether    *
*the word has or does not have an accent                   *
***********************************************************)
procedure process(word:wordType);
   type
      LetPosType   = 1 .. MaxChar;
      SylPosType   = 1.. MaxSyl;
      SylArray     = array[1..maxSyl] of string[SylSize];
      SoundType    = array[1..MaxChar] of boolean;
   var
      TempString: WordType;
      Syllables:SylArray;
      Vowel,
      Strong,
      Boundary       : SoundType;
      EndPlus,
      CountPlus      : boolean;
      LastSyl,
      StressSyl      : SylPosType;
      StressLet      : LetPosType;

(*********************************************
*The CheckNumeric procedure verifies validity*
*of numeric input in order to proevent a     *
*program crash on bad input                  *
*********************************************)
procedure CheckNumeric(var TempString:WordType);
   begin
     while not Numeric(TempString) do
        begin
           writeln('**** Garbage Input ***');
           write('Please enter a numeric value > ');
           readln(TempString)
        end
     end; (*CheckNumeric*)

(******************************************
*procedure CheckSyl verifies the validity *
*of the syllable number that is supposed  *
*to carry the stress.                     *
*Ie. It should not be 0 and not greater   *
*then the number of syllables             *
******************************************)
procedure CheckSyl( var StressSyl:integer);
   begin
```

```
   while not (StressSyl in [1..LastSyl]) do
      begin
         writeln('Invalid syllable number');
         write('Enter correct emphasis syllable >');
         readln(TempString);
         CheckNumeric(TempString);
         StressSyl:=Convert(TempString)
      end
   end; (*CheckSyl*)


(*********************************************************
*The Initialize procedure sets the variables to initial*
*values in preparation for a new word of input         *
*********************************************************)
procedure Initialize(var Strong,Boundary,vowel:SoundType;
                     var Syllable:SylArray);
   var
      i:integer;
   begin
      for i:=1 to MaxChar do
         begin
            boundary[i]:=false;
            vowel[i]:=false;
            Strong[i]:=false
         end;
      boundary[length(word)+1]:=true;
```

(Continued on next page)

Accent Finder Listing (Listing Continued, text begins on page 82)

```
      for i:=1 to MaxSyl do Syllable[i]:=''
   end; (*Initialize*)

(*********************************************
*procedure SetQuality sets the quality of each   *
*sound of the word. Ie. if a vowel then the vowel*
*flag is turned on and if a strong vowel then the*
*strong flag is turned.                          *
*********************************************)
procedure SetQuality(var Strong,vowel:SoundType);
   var
      i:integer;
   begin
      for i:=1 to length(word) do
         if word[i] in vowels
         then
            begin
               vowel[i]:=true;
               if word[i] in StrongVowels
               then Strong[i]:=true
            end
   end; (*SetQuality*)

(******************************************************
*the SetSylls procedure sets the boundary between the  *
*syllables of the word according to strength and value.*
******************************************************)
procedure SetSylls(var Boundary:SoundType; word:wordType;
                       vowel   :SoundType);
   var
      i:integer;
   begin
      for i:=2 to length(word) do
         begin
            if ((not vowel[i]) and vowel[i-1])
            or ((word[i-1] = 'N') and (word[i]='T'))
            or (strong[i-1] and (not strong[i]))
            or (strong[i-1] and strong[i])
            or (strong[i] and (not strong[i-1]) and vowel[i-1])
            then boundary[i]:=true;
            if i>2 then
            if (vowel[i-2] and (not vowel[i-1]) and (not vowel[i])
             and (word[i]=word[i-1]))
            then
               begin
                  boundary[i-1]:=true;
                  boundary[i]:=false
               end
         end
   end; (*SetSylls*)

(**************************************************************
*The SplitConsonants splits two neighbouring consonants in    *
*syllables (except ll)                                        *
**************************************************************)
```

```
procedure SplitConsonants(var boundary:SoundType; word:WordType;
                              vowel   :SoundType);
  var
    i:integer;
  begin
    for i:=3 to length(word) do
      if vowel[i-2] and (not vowel[i-1]) and (not vowel[i])
      then
        begin
          boundary[i-1]:=false;
          boundary[i]:=true
        end
  end; (*SplitConsonants*)
(**********************************************
*The WeldCH procedure insures that C and H go in  *
*the same syllable                                 *
**********************************************)
procedure WeldCH(var boundary: SoundType; word:wordType;
                     vowel    :SoundType);
  var
    i:integer;
  begin
    for i:=3 to length(word) do
      begin
        if (word[i-1]='C') and (not Boundary[i-1]) and
```

(Continued on next page)

Accent Finder Listing (Listing Continued, text begins on page 82)

```
          (word[i]='H') and boundary[i]
        then
          begin
            Boundary[i-1]:=true;
            Boundary[i]:=false
          end
      end
  end; (*WeldCH*)

(*****************************************************
*Liquids (l and r) must be spliced to the next       *
*syllable                                            *
*****************************************************)
procedure WeldLiquids(var boundary: Soundtype; word:wordType;
                          vowel   : SoundType);
  var
    i:integer;
  begin
    for i:=3 to length(word) do
      if vowel[i-2] and
        (not vowel[i-1]) and (not vowel[i]) and
        (word[i] in liquids)
      then
        begin
          boundary[i-1]:=true;
          boundary[i]:=false
        end
  end; (*WeldLiquids*)


(************************************************************
*The divide procedure sets the array of strings 'Syllables'*
*to the values of the word syllables by splicing together  *
*the letters of a word up to a letter marked as a syllable *
************************************************************)

procedure divide(boundary:SoundType; var Syllables:SylArray);
  var
    i,
    j:integer;
  begin
    i:=1;
    Syllables[1]:=copy(word,1,1);
    j:=2;
    while j<=length(word) do
      begin
        if not boundary[j]
        then Syllables[i]:=Syllables[i]+copy(word,j,1)
        else
          begin
            Syllables[i+1]:=copy(word,j,1);
            i:=i+1
          end;
        j:=j+1
      end
  end; (*divide*)
```

```
(***************************************************************
*The EvalSylls procedure displays the word split in syllables*
*and it also returns their total number.                      *
*The syllables are numbered.                                  *
***************************************************************)
procedure EvalSylls(syllables: SylArray; var LastSyl:integer);
   var
      i:integer;
   begin
      i:=1;
      while syllables[i]<>'' do
         begin
            write('***',i:1,'->',syllables[i]);
            i:=i+1
         end;
      LastSyl:=i-1
   end; (*EvalSyll*)

(*****************************************************
*The procedure SetStressLet will set the position   *
* of the stress letter by doing some calculations   *
* I.e. looking for the stressed vowel within the    *
* stressed syllable                                 *
*****************************************************)
procedure SetStressLet(var StressLet:integer);
```

(Continued on next page)

Accent Finder Listing (Listing Continued, text begins on page 82)

```
   var
      SyllCount,
      i           :integer;
   begin
      SyllCount:=1;
      for i:=1 to length(word) do
         begin
            if Boundary[i]
            then SyllCount:=SyllCount+1;
            if (SyllCount=StressSyl) and Vowel[i]
            then StressLet:=i
         end
   end; (*SetStressLet*)

(*********************************************
*procedure DisplayAccent displays the word   *
*with its written accent as a slash next to  *
*the letter supposed to carry it.            *
*********************************************)
procedure DisplayAccent(word:WordType);
   var
      i:integer;
   begin
      writeln('The word ',word,' has a written accent ');
      for i:=1 to length(word) do
         begin
            write(' ',word[i]);
            if i=StressLet then write('/')
            else write('  ')
         end;
      writeln
   end; (*DisplayAccent*)

(********************************************
procedure InfoMsg comes to work after all   *
procedures have done their work and it      *
explains why certain decisions were made    *
********************************************)

procedure InfoMsg(word: WordType; EndPlus, CountPlus: boolean);
   begin
      writeln('The final letter is ',word[length(word)]);
      write('Thus a ');
      if EndPlus then write('+')
      else write('-');
      writeln(' must be assigned for the word ending.');
      writeln('The emphasis is on syllable #',StressSyl:1);
      write('Thus we must assign ');
      if CountPlus then write('+')
      else write('-');
      writeln(' for the emphasis placement.');
      writeln('Reasons considered for the above decision: ');
      If EndPlus<>CountPlus
      then writeln('The signs are different so there is an accent!')
      else writeln('The signs are the same so there is no accent!');
      if (not strong[length(word)-1]) and vowel[length(word)-1] and
```

```
        Strong[Length(word)]
      then writeln('The word ends with a diphthong, there is an accent.')
      if strong[StressLet-1] and (not strong[StressLet])
      then writeln('A split diphtong, there is an accent.');
      if (word[length(word)-1]='P') and (word[length(word)]='S')
      then writeln('Words ending in PS always get accents!');
      if (length(word)=3) and (not vowel[1])
      then writeln('Three letter words are affected by semantics',
                   ' (Ex. mas vs. ma's)')
   end; (*InfoMsg*)

begin(*of the process procedure*)
   Initialize(Strong,Boundary,vowel,Syllables);
   SetQuality(strong,vowel);
   SetSylls(boundary,word,vowel);
   SplitConsonants(boundary,word,vowel);
   WeldCH(boundary,word,vowel);
   WeldLiquids(boundary,word,vowel);
   if not vowel[length(word)]
   then boundary[length(word)]:=false;
   divide(boundary,syllables);
   EvalSylls(syllables,LastSyl);
   write('Enter the number of emphasis syllable > ');
   readln(TempString);
   CheckNumeric(TempString);
   StressSyl:=Convert(TempString);
```

(Continued on next page)

```
   CheckSyl(StressSyl);
   SetStressLet(StressLet);
   EndPlus:=false;
   CountPlus:=false;
   if not ((not vowel[length(word)]) and (not vowel[length(word)-1]))
   then if (vowel[length(word)]) or
           (word[length(word)] in ['N','S'])
        then EndPlus:=true;
   if LastSyl-StressSyl=1 then CountPlus:=true;
   if (EndPlus<>CountPlus) or
      ((length(word) > 3) and
      strong[StressLet-1] and (not strong[StressLet])) or
      (((not strong[length(word)-1]) and Vowel[length(word)-1]) and
      Strong[length(word)])
   then DisplayAccent(word)
   else writeln('No accent !');
   InfoMsg(word,EndPlus,CountPlus)
end;  (*Process*)

begin (*main*)
   word:='xxx';
   vowels:=['A','E','I','O','U'];
   liquids:=['L','R'];
   StrongVowels:=['A','E','O'];
   LowerCaseSet:=['a'..'z'];
   Letters:=['A'..'Z'];
   writeln('Please remeber that all interrogatives get an accent');
   writeln('when used in a question.');
   writeln('Ex: ?Que'' dice?');
   writeln('    E''l dice que tiene once anos.');
   writeln('To stop enter ''Return'' without input');
   while word<>'' do
      begin
         Prompt(word,error);
         if (word<>'') and (not error)
         then
            begin
               if length(word) < 3 then
               case length(word) of
                  1: writeln('Monosounds do not have accents');
                  2: if word='SI' then SImsg
                     else if word='MI' then MImsg
                          else if word='EL' then ELmsg
                               else
                                  writeln('Probably no accent !')
               end
               else Process(word)
            end
         end;
      writeln('!!Que la fuerza este'' contigo!!')
   end.
```

End Listing

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 756 MENLO PARK, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Dr. Dobb's Journal
For the Experienced in Microcomputing

2464 EMBARCADERO WAY
PALO ALTO, CA 94303

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 756 MENLO PARK, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Dr. Dobb's Journal
For the Experienced in Microcomputing

2464 EMBARCADERO WAY
PALO ALTO, CA 94303

Dr. Dobb's Journal
For the Experienced in Microcomputing

Reader Service Card

To obtain information about products or services mentioned in this issue, circle the appropriate number listed below. Use bottom row to vote for best article in issue. This card is valid for 90 days from issue date. Use only one card per person.

May 1984, No. 91

Name ______________________

Address ______________________

City/State/Zip ______________________ Phone (   ) ______________

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81
82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108
109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135
Articles:	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	

Comments:

Purchase of Magazine:

1 ☐ Subscription
2 ☐ Computer Store
3 ☐ Newsstand
4 ☐ Bookstore
5 ☐ Passed on by friend/colleague
6 ☐ Other ______________

Dr. Dobb's Journal
For the Experienced in Microcomputing

Reader Service Card

To obtain information about products or services mentioned in this issue, circle the appropriate number listed below. Use bottom row to vote for best article in issue. This card is valid for 90 days from issue date. Use only one card per person.

May 1984, No. 91

Name ______________________

Address ______________________

City/State/Zip ______________________ Phone (   ) ______________

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81
82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108
109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135
Articles:	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	

Comments:

Purchase of Magazine:

1 ☐ Subscription
2 ☐ Computer Store
3 ☐ Newsstand
4 ☐ Bookstore
5 ☐ Passed on by friend/colleague
6 ☐ Other ______________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 756 MENLO PARK, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Dr. Dobb's Journal
For the Experienced in Microcomputing

2464 EMBARCADERO WAY
PALO ALTO, CA 94303

Dr. Dobb's Journal
NOW AT BIGGER SAVINGS!

~~$35.40~~ $25.00

If you take advantage of this special offer you save over $10 off newsstand prices, that's a 30% savings!

_____Please charge my: ___Visa ___MasterCard ___American Express
_____Payment enclosed _____Bill me later

Card # ________________ Exp. date ________

Signature ________________

Name ________________

Address ________________

City __________ State _____ Zip ______ V4

Offer good in USA only. Foreign rates upon request. Please allow up to six weeks for first issue.
This offer good until August 31, 1984.

A publication of People's Computer Co., P.O. Box E, Menlo Park, CA 94026.

Dear Reader, **May 1984, No. 91**

Dr. Dobb's has a long tradition of listening to its readers. We like to hear when something really helps or, for that matter, bothers you. In this hectic world of ours, however, it is often difficult to take the time to write a letter. This card provides you with a quick and easy way to correspond. Simply fill it out and drop it in the mail. We take care of the rest. Thanks for taking a few minutes to talk with us.

—Ed.

Which articles or departments did you enjoy the most this month? Why? (Please indicate order of preference.)

____

____

Comments or suggestions: ________________

____

____

Name ________________

Address ________________

____

DDJ BACK ISSUES

#66 Volume VII, No. 4: 8080-Z80/8086 Cross-Assembler, Part 2 – Writing the Runic Compiler – Poor Person's Spelling Checker.

#68 Volume VII, No. 6: Multi-68000 Personal Computer – PDP-1802, Part One – Improved LET for LLL Basic – CP/M Print Utility.

#69 Volume VII, No. 7: IBM-PC Issue! CP/M-86 vs. MSDOS (A Technical Comparison) – Hi-Res Graphics on the IBM-PC – PDP-1802, Part II – Review of Word Processors for IBM.

#70 Volume VII, No. 8: Argum "C" Command Line Processor – SEND/RECEIVE File Transfer Utilities – Intel's 8087 – Performance Evaluation.

#71 Volume VII, No. 9: FORTH Issue! Floating-Point Package – H-19 Screen Editor – Relocating, Linking Loader – Z8000 Forth – Forth Programming Style – 8086 ASCII-Binary Conversion Routines – CP/M Conditional SUBMIT.

#72 Volume VII, No. 10: Portable Pidgin for Z80 – 68000 Cross Assembler, Part I – MODEM and RCP/Ms – Simplified 68000 Mnemonics – Nested Submits – 8086/88 Trig Lookup.

#73 Volume VII, No. 11: Wildcard UNIX Filenames – Tests for Pidgin – 68000 Cross Assembler Listing, Part 2 – Adding More BDOS Calls – The Perfect Hash – BASIC Memory Management – Benchmarks for CP/M-86 vs. MSDOS, and the 8087.

#75 Volume VIII, No. 1: Augusta, An ADA Subset for Micros – Xanadu: Hypertext from the Future – Stone Age Computers: 6000 Years of Computing Science – Small-C Compiler v2., Part 2.

#76 Volume VIII, Issue 2: PISTOL, A Forth-like Portably Implemented STack Oriented Language – Program Linkage by Coroutines, Forth to BASIC – Linking CP/M Functions to Your High-Level Program – Concurrent CP/M-86 – Small Systems Engineering CP/M-80 Expansion Card for the Victor 9000 – REVAS Disassembler.

#77 Volume VIII, Issue 3: Augusta, Part II: The Augusta P-Code Interpreter – A Small-C Operating System – 6809 Threaded Code: Parametrization and Transfer of Control – A Common-Sense Guide to Faster, Small BASIC – A Fundamental Mistake in Compiler Design – Basic Disk I/O, Part I.

#78 Volume VIII, Issue 4: RECLAIM Destroyed Directories – Binary Magic Numbers – 8080 Fig-Forth Directory & File System – SAY" Forth Votrax Driver – TRS-80 8080 to Z80 Translator – Basic Disk I/O, Part II.

#79 Volume VIII, No. 5: Augusta Part III: The Augusta Compiler – A Fast Circle Routine – Enhancing the C Screen Editor – Shifts and Rotations on the Z80 – The SCB, TSX, and TXS Instructions of the 6502 and 6800 – MS-DOS vs. CP/M-86 – Controlling MBASIC – The Buffered Keyboard – IBM PC Character Set Linker – Flip Utility for the IBM PC.

#80 Volume VIII, Issue 6: Fast Divisibility Algorithms – B-Tree ISAM Concepts – CP/M BDOS and BIOS Calls for C – Serial Expansion in Forth – Fast Matrix Operations in Forth, Part I – Yes, You Can Trace Through BDOS – Julian Dates for Microcomputers – 8088 Addressing Modes – 8088 Line Generator – CP/M Plus.

#81 Volume VIII, Issue 7: Augusta, Part IV (The Augusta Compiler, continued) – RED: A Better Screen Editor, Part I – Anatomy of a Digital Vector and Curve Generator – Fast Matrix Operations in Forth, Part II – The AGGHHH! Program – MBOOT Revisited – CP/M Plus Feedback – MS-DOS Rebuttal – 68000 Tools – Sizing Memory on the IBM PC.

#82 Volume VIII, Issue 8: Serial-to-Parallel: A Flexible Utility Box – McWORDER: A Tiny Text Editor – And Still More Fifth Generation Computers! – Specialist Symbols and I/O Benchmarks for CP/M Plus – CP/M Plus Memory Management – Zero Length File Test – PAUSEIF, QUITIF, and now SKIPIF – ACTxx Cross Assemblers.

#83 Volume VIII, No. 9: FORTH ISSUE! Forth and the Motorola 68000 – Nondeterministic Control Words in Forth – A 68000 Forth Assembler – GO in Forth – Precompiled Forth Modules – Signed Integer Division – Some Forth Coding Standards – The Forth Sort – A speed and accuracy benchmark program for high-level languages – Using a Digital Spreadsheet Program for Something Fun and Unusual.

#84 Volume VIII, No. 10: DDJ's new C/Unix column! – Unix to CP/M Floppy Disk File Conversion – A Small-C Help Facility – Attaching a Winchester Hard Disk to the S-100 Bus – Using Epson Bit-Plot Graphics – Your Fantasy Computer System – 8086/88 Function Macros – Auto Disk Format Selection – CP/M Plus Device Tables.

#85 Volume VIII, Issue 11: A Kernel for the MC68000 – A DML Parser – Towards a More Writable Forth Syntax – Simple Graphics for Printer – 8080 to Z80 Program Conversion, CP/M Plus DPB Macro Fix, and Quicker Submit File Truncation – Floating-Point Benchmarks and an MSDOS COM File Loader – software and book reviews.

#86 Volume VIII, Issue 12: Faster Circles for Apples – Cursor Control for Dumb Terminals – Dysan's Digital Diagnostic Diskette – Building a Programmable Frequency Synthesizer – Unix and *Non*-Interactive, User-*Un*friendly Software – Interfacing a Hard Disk Within a CP/M Environment – The New MS-DOS EXEC Function.

#87 Volume IX, Issue 1: NBASIC: A Structured Preprocessor for MBASIC – A Simple Window Package – Forth to PC-DOS Interface – Sorted Diskette Directory Listing for the IBM PC – Emulate Word-Star on TOPS-20 – More on optimising compilers – Problems under CP/M Plus with PAUSE – The PIP mystery device contest – Microsoft BASIC – An improved CLINK utility.

#88 Volume IX, Issue 2: Telecommunications Issue! Micro to Mainframe Connection – Communications Protocols – Unix to Unix Network Utilities – VPC: A Virtual Personal Computer for Networks – PABX and the Personal Computer – BASIC Language Telecommunications Programming – U.S Robotics S-100 Card Modem – Sysdrive program, bringing up CP/M Plus – PCDOS Close Function – The Microsoft Assembler – more on C layout standards – book and software reviews.

#89 Vol. 9, Issue 3: RSA: A Public Key Cryptography System, Part I – Introduction to PL/C: Programming Language for Compilers – Program Design Using Pseudocode – More on Binary Magic Numbers – Basically Precise – Stocking Up – How fast is CP/M Plus? – CP/M 2.2 BIOS Function: SELDSK – The results of the Floating-Point benchmark – book and software reviews.

#90 Vol. 9, Issue 4: Optimizing Strings in C – Expert Systems and the Weather – RSA: A Public Key Cryptography System, Part II – BASICFMT for TRS-80 – Several items on CP/M Plus – CP/M v2.2 Compatibility – BDOS Function 10: Vastly Improved – More on MS-DOS EXEC Function – MS-DOS Volume Labels – Finding Size of TPA under MS-DOS 2.0 – some comments about Unix – Low-Level Input-Output in C – More on Link Formats and Runtime Libraries – some information on TEX – software reviews.

TO ORDER:

Send $3.50 per issue (or $12.50 for every set of 5) to: Dr. Dobb's Journal, P. O. Box E, Menlo Park, CA 94026.

Please send me the issue(s) circled: **66 68 69 70 71**

72 73 75 76 77 78 79 80 81

82 83 84 85 86 87 88 89 90

I enclose $ ________ (U.S. check or money order).
Outside the U.S., add $.50 per issue.

Please charge my: ☐ Visa ☐ M/C ☐ Amer. Exp.

Card No. ______________________

Exp. Date ________

Signature ______________________

Name ______________________

Address ______________________

City ______________________

State ______________ Zip ________

Availability on first come/first serve basis. Outside the U.S., add $.50 per issue ordered. Price includes issue, handling, and shipment by second class or foreign surface mail. Within the U.S., please allow 6-9 weeks to process your order second class. For faster service within the U.S., we'll ship UPS if you *add* $1.00 for 1-2 issues and $.50 for *each issue* thereafter. We need a street address, not a P.O. Box. Outside the U.S., *add* $1.50 *per issue* requested for airmail shipment.

Circle **no. 83** on reader service card.

Solutions to Quirks in dBASE II

At Money Tree Software in Corvallis, Oregon, we offer a service to financial planners called MoneyNet. MoneyNet is a computer data base accessed by a modem from which planners can retrieve timely data about various financial matters.

The entire system is written in dBASE II. We tried several of the dBASE II look-alikes, which claim more records, more fields, and more memory variables, before committing to dBASE II. Let me say, with firsthand experience, that from a high-level language perspective dBASE II is the clear choice of most serious programmers.

However, as good as dBASE II is, it does have some quirks.

The quirk I will address here is the lack of an INKEY function like that found in other high-level languages. The INKEY can be used to interrupt a long listing or print-out without aborting the program entirely.

The Problem

Almost any standard CP/M text file can be converted to a dBASE II data file simply by creating a data base of one field, character type, and length of 80 then appending from the desired text file using the SDF option. Each record of the data base will be one line of text. Displaying the text can be a function of a normal DO-LOOP like this:

```
USE TEXTFILE
DO WHILE .NOT. EOF
? LINE
SKIP
ENDDO
```

But suppose this routine is part of a menu-driven, end-user application and the user decides in the middle of the DO-LOOP that he or she wants to stop reading this text and return to the main menu. How can the user abort the loop from the keyboard? ESCAPE has been turned off, since we don't want the end user at the DOT-PROMPT level, and it wouldn't meet the user's need anyway since he or she wants program flow to continue after exiting the DO-LOOP.

by Gene Head

Gene Head, 2860 NW Skyline Dr., Corvallis, OR 97330.

We need a way to check whether a key has been pressed at the keyboard after each record (one line of text) has been displayed. That way, our DO-LOOP could be aborted with a control-C from the keyboard as shown in Figure 1 (page 99).

The standard answer is to write a CALLable machine language routine to monitor the keyboard, usually by doing a BIOS function call. Unfortunately, this scheme will not work because dBASE II polls the keyboard at least once after a character is sent to the console, gobbling up the keyed-in character before any machine language call can grab it. No matter where in the program we put our call to console I/O check, it is very unlikely that our subroutine will recognize the user's key press before dBASE II intercepts it.

A Workable Solution

A disassembly examination of the dBASE II program shows that console input goes something like Figure 2 (page 99). It looks like the solution could be a simple matter of PEEKing at SAVEIT to see the last key pressed. Unfortunately, dBASE II zeros this byte after it uses the keyed-in data. This routine is called *very often*, and SAVEIT will be zero most of the time, containing the keyed-in character for only a very short period. My modification saves the keyed-in byte to a safe RAM location undisturbed by dBASE II (Figure 3, page 99).

This modification means that SAVEIT continues to perform as expected, but LASTKEY is filled with the keyed-in character and remains unchanged until another key is pressed or it is changed with a POKE command. Now we can monitor this location, much like BASIC allows for the INKEY command. The DO-LOOP will print text until the user inputs a control-C (or any desired character) from the keyboard or the end of file is found (Figure 4, page 99).

Of course, it isn't quite as easy as it sounds because there is no room to do a

Solutions to dBASE II Listing

```
A>DDT DBASE.COM
DDT VERS 2.2
NEXT PC
4E00 0100
-L 3A81
   3A81    MVI    E,FF
   3A83    MVI    C,06
   3A85    CALL   0005
   3A88    ORA    A
   3A89    RZ
   3A8A    STA    4378      <--- PATCH JMP HERE
   3A8D    RET
   3A8E    PUSH   B
   3A8F    PUSH   D
   3A90    PUSH   H
   3A91    CALL   3A62
-
-A3A8A
3A8A  JMP  14A              <--- PATCH JMP HERE
3A8D
-
-A14A
014A  STA  4378             <--- INSTRUCTION WE PATCHED OUT
014D  STA  151              <--- ADDED INSTRUCTION
0150  RET
0151  DB  0                 <--- 337 DECIMAL (LASTKEY)
0152
-
-GO
A>SAVE 77 DBASE.COM
A>
```

End Listing

```
USE TEXTFILE
DO WHILE .NOT. EOF .AND. KEY:PRESS<>3          * 3=CONTROL-3
? LINE
SKIP
ENDDO
```

Figure 1.

```
CONIN  MVI    E,OFFH    ; SET UP FOR DIRECT CONSOLE INPUT
       MVI    C,9
       CALL   BIOS      ; DO DIRECT CONSOLE I/O
       ORA    A
       RZ               ; NOTHING TYPED SO RETURN
       STA    SAVEIT    ; SAVE TYPED KEY
       RET
```

Figure 2.

```
CONIN  MVI    E,OFFH    ; SET UP FOR DIRECT CONSOLE INPUT
       MVI    C,9
       CALL   BIOS      ; DO DIRECT CONSOLE I/O
       ORA    A
       RZ
       STA    SAVEIT    ; SAVE keyed-in character
       STA    LASTKEY   ; . . . and to a SAFE location
       RET
```

Figure 3.

```
USE TEXTFILE
DO WHILE .NOT. EOF .AND. PEEK(LASTKEY) <> 3   * 3=CONTROL-C
? LINE
SKIP
ENDDO
```

Figure 4.

```
? 'YOU HAVE FIVE SECONDS TO ANSWER  --> '
STORE 100 TO TIMER
POKE 0,337
DO WHILE TIMER .AND. PEEK (337) = 0
STORE TIMER -1 TO TIMER
STORE PEEK (337) TO ANSWERED
ENDDO
IF ANSWERED
? 'YOU ENTERED AN ANSWER IN TIME'
ELSE
? 'YOU FAILED TO ENTER AN ANSWER IN TIME'
ENDIF
```

Figure 5.

direct in-line patch in dBASE II and my modification adds three bytes of code. So we have to do some redirection to an unused RAM area in dBASE II.

The screen I/O, as defined using the INSTALL.COM program, is saved in low memory from about 10D hex to about 152 hex. Because the codes starting at 14A hex are all zeros in my installed dBASE II, this is where I put the patch. As far as I can tell, dBASE II never addresses this RAM. *Your installed dBASE II may use these bytes*, so first look around this area for eight consecutive bytes of zeros and then put your patch there.

This is what will happen. Instead of storing the keyed-in character at SAVEIT using the STA SAVEIT instruction, we use these three bytes to do a jump to our unused area of RAM in the screen data table. Here we do the first store just like the store we patched over: STA SAVEIT. In addition, we save the keyed-in data to a safe RAM location with STA LASTKEY. Finally, we return. Our patch is done and dBASE II will not even know it is there!

If you use dBASE II 2.4, the DDT session in the Listing (page 98) will permanently modify your dBASE II to recognize a keyed-in character by PEEK (337). Now a PEEK (337) will always show the last key pressed. This can be used to timeout input, as shown in Figure 5 (below).

The DO-LOOP application has already been explained, and I hope you can come up with unusual applications I haven't thought of.

Next time I will present a short piece on verifying zip codes in mailing lists.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 197.**

OF INTEREST

by Michael Wiesenberg

But Does It Have "Of Interest"?

Want to find all articles about your IMSAI, Sorcerer, or HP 87? The **Microindex**, from Serious Personal Computing, is various flavors of periodicals that index the multiplicity of articles in over 70 computer magazines, a sort of Readers' Guide to Periodical Literature of computer publications. *Microindex*, a monthly for libraries, universities, and large businesses, cross-references the entire spectrum of computer magazines, and costs $99 a year or $12 an issue. The monthly *Abridged Microindex*, for smaller organizations, indexes a subset of 32, at $49 per year or $6 per issue. Yes, *DDJ* is in it. Annual *Microindexes* specific to various magazines cost $5 to $12 each. I know that you will all want to order the *Microindex* to *Dr. Dobb's Journal*, Volume 8, for $7. Also available are complete one-volume monthlies and annuals on various topics, as education, business, Apple, IBM, Commodore, and so on. **Reader Service No. 111.**

C You on 8086

Version 2 of **Lattice C** is now available for 8086 and 8088 for most MS-DOS computers. Programs can address 1Mb of memory, and be as long as 1Mb. The new library handles multilevel file directories and is compatible with recent versions of Unix. Compiler costs $500, and library source is also $500. You can get an update from whoever supplied your original. **Reader Service No. 109.**

Million Dollar Computer Poker Challenge

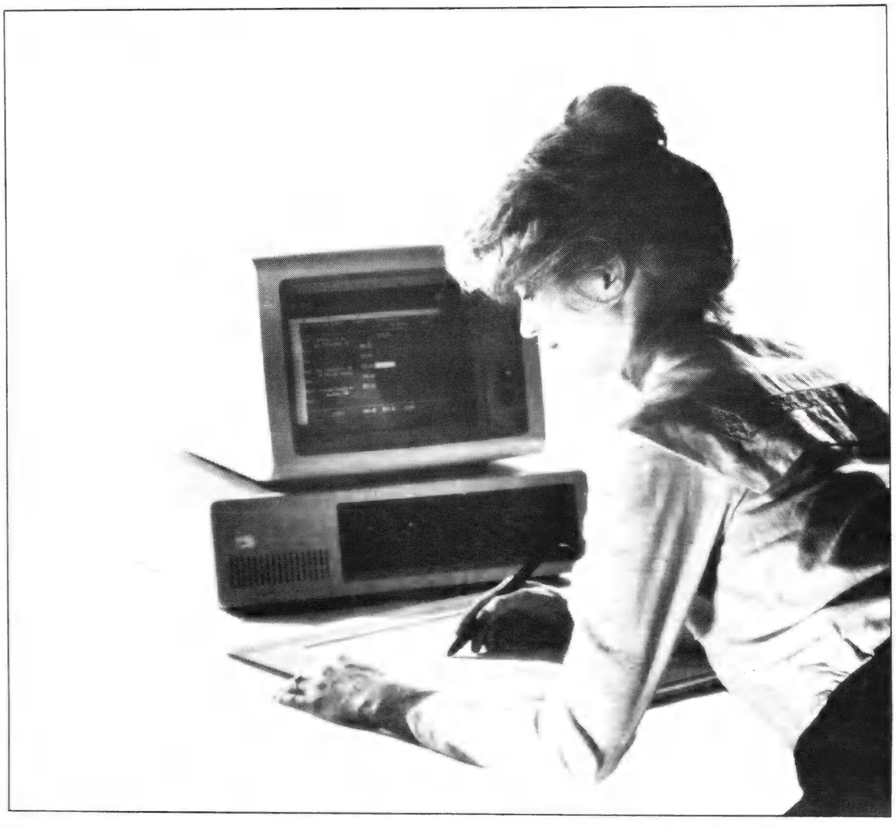
Can a computer play intelligent poker? You wouldn't think so to judge from the poker games available on today's micros, but **Mike Caro** ("The Mad Genius of Poker"), renowned poker writer and games programmer, is so convinced his Apple can beat any human that he has issued a $100,000 challenge to several world-class poker players, and been accepted by two former champions of **The World Series of Poker**. A Vegas casino owner has offered a counterchallenge: to play for *one million dollars cash*. To ensure that the computer is not "helped" in its decisions by the operator, special decks of cards with optical codes on the faces and optical scanners will be provided by the **Data Bar Corporation**. Think you can play poker better than a computer? You might get a chance to try if you've got a million dollars to back you up. Better yet, can you program your computer to beat Caro's? Win or lose, the publicity for your game would be worth well over a million dollars.

Caro also plans on releasing the challenge soon as a home game, with the slogan, "Play the program that won a million dollars in the World Series of Poker." I've seen an advance copy, and the card graphics are impressive, much better than those tiny attempts at exact representation that are impossible to read that I've seen in other commercial games. **Reader Service No. 101.**

But What if Your Printing Isn't Legible?

For those who get tired of using keyboards, Pencept's **Penpad** lets you input data by handwriting. I mean hand *printing*, and you'll have to learn to represent lowercase letters with half-height capitals, and you have to stay within the lines just like you did in first grade. The real value in this product seems to be in designing custom forms, in conjunction with the **Penpatch** utility.

Penpad is directly compatible with VisiCalc, MultiPlan, Lotus 1-2-3, and WordStar. For $995 on the PC, you get tablet, ballpoint with side-mounted activation button, and a board for the PC. The board has its own MC68K CPU that has the character recognition software, sending input that looks to the computer like it came from the keyboard. Penpad is also what they call "the mouse that writes," because stylus motion and button pressing can be translated into cursor commands. **Reader Service No. 115.**



Save A Sorcerer

Now that Exidy no longer makes them, how do you thousands of Sorcerer owners get service and answers to your questions? Contact **Jack MacGrath,** who provides technical support, has parts, services Micropolis drives, and buys and sells systems. **Reader Service No. 103.**

Less for 40

You may recall my predicting a year ago that prices would come down considerably on hard disks with increased storage. CompuPro's **H40 Hard Disk Subsystem** for its IEEE 696/S-100 bus compatible microcomputers costs $5495, and comes with a 40Mb Quantum 5.25-inch hard disk, CompuPro's Disk 3 DMA disk controller, CP/M-80 or -86, and a double-density Qume Trak 842 floppy disk drive with 2.8Mb for backup on single- or double-density, single- or double-sided media.

And, while they've got our attention, CompuPro would like us to know that they have the first commercially available IEEE 696-compatible, 80286-based, multiuser microcomputer. This is, of course, Intel's new CPU that is upward compatible from the 8088 and 8086, but runs at 6 MHz, or twice as fast as 8086- or 68K-based systems. The **System 816/F** also has an 80287 math coprocessor. It comes with 512K 16-bit static memory, expandable to 1Mb, 12 serial ports, one Centronics-compatible printer port, one parallel port, 1.2Mb floppy storage, a 40Mb hard disk, and CP/M-86 and MP/M-86. $14,995 walks away with it. **Reader Service No. 119.**

Or Pascal

Limbic Systems has a **Pascal Compiler** for the Commodore 64 that generates native code. The package comes with linker, debugging facilities, and editor, and costs a phenomenal $50. **Reader Service No. 107.**

Zenith Pretends It's PC

Z-Util from Lindley Systems is IBM PC emulation for the Zenith Z-100, emulating keyboard input, screen printing and scrolling, and printer output, so that WordStar, the Perfect series, and dBaseII all run. Only modem and graphics programs don't run, because of the differences in hardware. Screen dumps, however, do work, with automatic logic-seeking and correct screen aspect ratios. All this for $35. **Reader Service No. 121.**

New Character for Old Printwheels

How often have you wished for an odd symbol on your printer: an at-sign, or copyright symbol, for instance, or maybe that elusive ™ that actually exists on some printwheels? **Business Support Services** will replace any character on your Diablo, Xerox, or Qume metal or plastic printwheel with any other available character. **Reader Service No. 117.**

64-4th or Fight

C64 UNIFORTH, from Unified Software Systems for the Commodore 64, is they claim, "*virtually* identical to our other FORTH systems, making application software particularly easy to port." The emphasis is mine, and the reason it's there is because *virtually* is one of those all-inclusive advertising words that means whatever the person using it wants it to. As Humpty Dumpty said, "When I make a word do a lot of work like that, I always pay it extra." That is, as Sportin' Life sang, "It ain't necessarily so." I'm not saying their claim isn't valid, just that *virtually* is one of those words that doesn't like to be pinned down. (Neither does *compatible*.) Anyway, high-level source files use what they call the 1541 "relative file" access, with words included to create, delete, switch and load from these files. You can set and read directly from UNIFORTH the system clock time and date, access the graphics controller chip, including analog-to-digital conversion, and control the light pen, joystick, and three sound voices. You get a complete 6502 assembler, with easy data stack accessing macros, and a video editor with cursor addressing. The integer version, $70, is available now, while the software floating-point version, $125, is coming soon. You can upgrade for the difference in price. Full kernel sources cost $90. Write for a free brochure. **Reader Service No. 105.**

Catch Your Words

You can get an inexpensive printout basket on which your printer sits from **Systems Enhancement Engineering.** $22.50 for the 12-inch model and $24.50 for 18, plus $3 p. and h. **Reader Service No. 113.**

Not Another Cute Head!

Tweaker, from Kludge Enterprises, is a hardware/software product for programmers and hardware engineers that puts that extra little bit into products always needed to ship them out the door. Specify format. $595 FOB VARPTR LAX. **Reader Service No. 0.**

Contact Points

Business Support Services, Inc., 705 Butternut Ave., Royal Oak, MI 48073; (313) 585-4736.

Caro's Poker Challenge: contact Henri Bollinger Agency, 9200 West Sunset Boulevard, Los Angeles, CA 90038; (213) 274-8483.

CompuPro, 3506 Breakwater Court, Hayward, CA 94545; (415) 786-0909.

Data Bar Corporation, 10202 Crosstown Circle, Eden Prairie, MN 55344; (800) 672-2776.

Lattice, Inc., Box 3072, Glen Ellyn, IL 60138; (312) 858-7950.

Limbic Systems, 560 San Antonio Road, Suite 202, Palo Alto, CA 94306; (415) 424-0168.

Lindley Systems, 21 Hancock Street, Bedford, MA 01730; (617) 275-6821 (evenings and weekends).

Jack MacGrath, 73 Jordan Road, North Chelmsford, MA 01863; (617) 251-4776.

Pencept, Inc., 39 Green St., Waltham, MA 02154; (617) 893-6390.

Serious Personal Computing, Box 7059, South Nashua, NH 03060; (603) 888-1376.

Systems Enhancement Engineering, Box 40215, Indianapolis, IN 46240; (317) 844-8817.

Unified Software Systems, Box 2644, New Carrollton, MD 20784; (301) 552-9590.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 198.**

ADVERTISER INDEX


INFINITE
SOLUTIONS
Like the molecules in a snowflake, the elements of a computer database can be structured and related in an infinite variety of combinations. Being able to present these combinations quickly and efficiently, with maximum flexibility and minimum programming knowledge, is the mark of excellence which sets the sophisticated database management system apart from the ordinary.
Thousands of satisfied users, from amateur and professional programmers to government agencies, major corporations and industries, have found that DataFlex has no equal in applications development software.
Write or phone for our latest literature and a list of existing applications.
Compatible with CP/M, CP/M-86, MP/M-86, MSDOS, PCDOS, TurboDOS, Novell Sharenet, PC-Net, Molecular N-Star, Televideo MmmOST, Action DPC/OS, Omninet, IBM PC w/Corvus and OSM Muse.
DATAFLEX™
APPLICATIONS DEVELOPMENT SOFTWARE
DATA ACCESS CORPORATION
8525 S.W. 129th Terrace, Miami, Florida 33156 (305) 238-0012 TELEX 469021 Data Access CI
MSDOS is a trademark of Microsoft. CP/M and MP/M are trademarks of Digital Research. DataFlex is a trademark of Data Access Corp.
Circle no. 21 on reader service card.